HZ BOOKS 华章教育
PEARSON

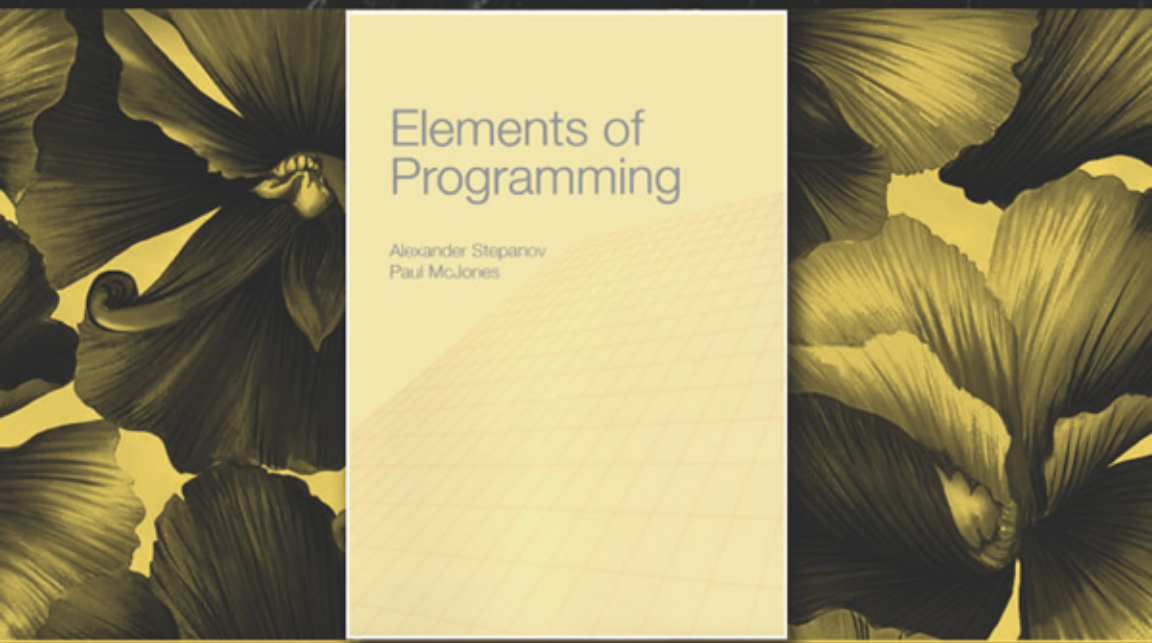
计 算 机 科 学 丛 书

追溯数学原理 探求编程原本
STL之父力作，C++之父鼎力推荐
北大数学学院裘宗燕教授倾情献译

编程原本

（美）Alexander Stepanov Paul McJones 著 裘宗燕 译

Elements of Programming



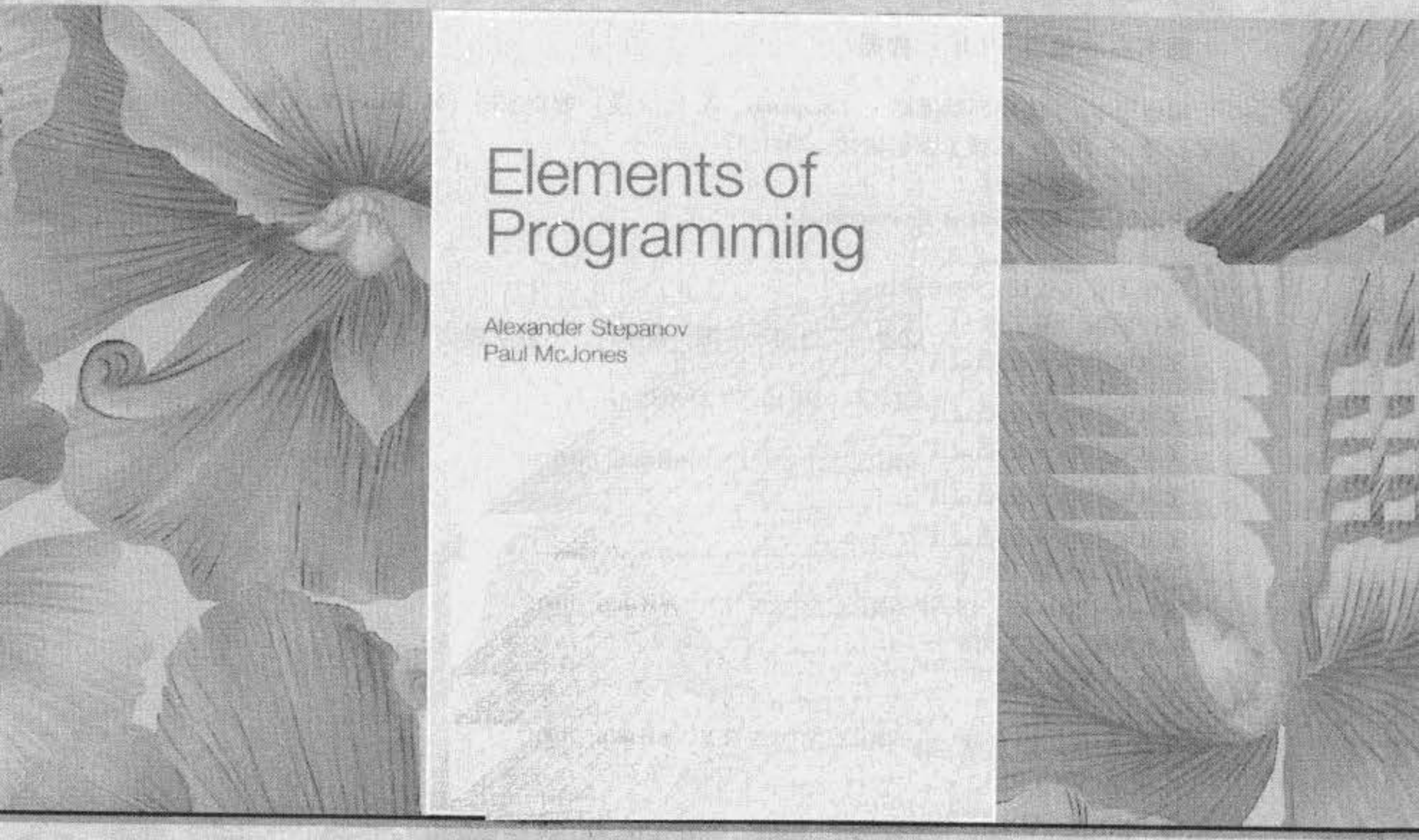
机械工业出版社
China Machine Press

计 算 机 科 学 丛 书

编程原本

（美）Alexander Stepanov　Paul McJones 著　裘宗燕 译

Elements of Programming



机械工业出版社
China Machine Press

本书将严格的数学定义、公理化和演绎方法应用于程序设计，讨论程序与保证它们能正确工作的抽象数学理论之间的联系。书中把理论的规程、基于这些写出的算法，以及描述算法性质的引理和定理一起呈现给读者，以帮助我们将复杂系统分解为一些具有特定行为的组件。

本书适合软件开发人员和需要进行程序设计的科学家及工程师阅读，也可供高等院校计算机及相关专业的师生参考。

本书版权登记号：图字：01-2010-1034

图书在版编目（CIP）数据

编程原本/（美）斯特潘诺夫（Stepanov, A.），（美）麦克琼斯（McJones, P.）著；裘宗燕译. —北京：机械工业出版社，2011.12
（计算机科学丛书）
书名原文：Elements of Programming

ISBN 978-7-111-36729-1

Ⅰ. 编…　Ⅱ. ①斯…　②麦…　③裘…　Ⅲ. 程序设计－数学理论　Ⅳ. TP311.1

中国版本图书馆 CIP 数据核字（2011）第 250685 号

机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码　100037）
责任编辑：关　敏
北京瑞德印刷有限公司印刷
2012 年 1 月第 1 版第 1 次印刷
185mm×260mm · 18.5 印张
标准书号：ISBN 978-7-111-36729-1
定价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991；88361066
购书热线：（010）68326294；88379649；68995259
投稿热线：（010）88379604
读者信箱：hzjsj@hzbook.com

译者序

在Addison-Wesley新书预告上看到本书时，我就感觉到本书的不同凡响，觉得应该让更多国内同行了解书中蕴涵的理念. 我立刻把信息告诉了华章的朋友. 没多久就得到了华章引进版权的消息.

市面上讨论编程的书籍浩如烟海. 说起编程，人们头脑中浮现的多半是语言、代码、hacking、测试、排除程序错误，以及与之相关的许多琐碎事务. 而本书作者看到的重点却不同. 在讨论编程时，他们关注的是数学、结构、规律、规范性、抽象、推导、前后条件、验证等等. 本书作者的技术水平和成就毋庸置疑，但为什么他们能构造出像STL那样的巅峰之作，这件事却值得认真思考. 如果国内有人说程序的基础是数学，估计会有不少人对其嗤之以鼻："你懂得什么是程序？写过多少行代码？" 但是，恐怕无人敢小觑本书作者，其见解和论据也无法忽视. 基于上述基本考虑，本书中给出了大量精妙而且根基坚实的程序，解决了一个个具体而重要的问题. 进一步说，作者还揭示了这些程序的理论基础，并从多个角度建立起它们之间的联系，使之可能成为无数实际程序的基础构件. 作者的根本目标，或许就是希望基于这种思维方法和开发技术，为范围广泛的软件系统建立起坚实基础. 在这里看不到调侃和讨好读者的流行俗语或插科打诨，只有严肃的叙述、分析和讨论. 阅读本书的过程绝不会轻松，但我们可以相信，在这里的付出会使人收获厚丰.

作者取"Elements"作为书名也很值得玩味. 我们都知道欧几里得的名著"Στοιχεία" (英译"Elements")，中文译为《几何原本》可能偏离了作者的本意. 欧氏应该是想为一种世界观和方法论提供一个范本，其内涵和意义绝不限于今天所说的"几何"领域. 实际上，《原本》希望展示的是思维和研究背后的一套基础概念和思想. 现在我们面对的是程序，这是一个完全人造的世界，这里的一切也有什么"本原"吗？本书作者基于其经验和认识也想来尝试一下，模

仿欧几里得探究一下位于所有编程背后的最根本的东西. 作者认为, 程序背后的本原就是数学的各种概念、技术和方法, 需要演绎、推导和证明等, 而绝不是模糊的想法和草率的编码和蛮力调试. 随着计算机被更广泛地应用于各种重要领域, 只靠朴素认识去工作将越来越显得脆弱和不可信. 从这个角度看, 本书也可以看成是作者的规劝和忠告.

我本无意翻译本书, 但经不住华章的朋友一再相邀, 只好勉强为之. 原书出自作者上课的幻灯片, 言简意赅, 涉及面广, 找到简洁又切中原义的译文常常很难, 一句译文经常需要斟酌很久, 每一遍检查都会发现许多不如意之处. 由于没有足够的时间, 经常有其他事务干扰, 这个翻译工作一拖再拖. 但无论如何, 一件事总要有个结束. 我决定现在将此工作告一段落, 将结果交给出版社和读者检验. 我希望这个中译本能对读者有所帮助, 也为其中的缺陷和不足负责.

为尽可能保持原貌, 我基于原书的 LaTeX 文件编辑译文 (感谢作者提供的原文件), 写了许多 LaTeX 宏定义, 最后用 MiKTeX 生成全书的 PDF 文件交给出版社. 这样做多花了不少时间, 但有利于保持全书 (及与原书) 的统一性, 可以减少不应出现的小错误. 当然, 这种费时费力的做法也使排版错误变成了我的个人责任. 本译本已收入原作者迄今的所有勘误, 翻译中发现的原书错误也都得到了作者确认. 为方便读者阅读, 本书实际上提供了两套索引: 原书的英文索引都予以保留, 与之对应, 我又加入了一套中文索引 (通过 LaTeX 生成), 供读者交叉参考. 原作者自创了不少术语, 没有习见译法, 我只能设法编造出相应译文, 其合理性需要时间检验. 此外, 正文中出现的术语都给出了英文对照.

有关本书的工作, 我要感谢刘海洋在使用 LaTeX 处理中文方面的若干意见, 感谢编辑发现了译稿的一些文字错误.

裘宗燕

2011年10月6日, 于北京大学理科一号楼

前　言

本书将演绎方法应用于程序设计，讨论程序与保证它们能正确工作的抽象数学理论之间的联系. 书中把反映这些理论的规程 (specification)，基于这些理论写出的算法，以及描述算法性质的引理和定理一起呈现给读者. 这些算法在一种实际程序设计语言里的实现是本书的中心. 虽然规程主要是供人阅读，但它们也应该 (或者说必须) 严格地与非形式化的、供机器使用的代码相结合，必须在通用的同时又是抽象而且精确的.

与在其他科学和工程领域里的情况一样，适合作为程序设计的基础的同样是演绎方法. 演绎方法能帮助我们将复杂系统分解为一些具有特定数学行为的组件，而这种分解又是设计高效、可靠、安全和经济的软件的前提.

本书是想奉献给那些希望更深入地理解程序设计的人们，无论他们是专职软件开发人员，还是把程序设计看作其专业活动中一个重要组成部分的科学家或工程师.

本书编写的基本想法是让读者从头到尾完整阅读. 读者只有通过阅读代码、证明引理、完成练习，才能真正理解书中的各方面材料. 此外我们还建议了一些项目，其中有些是完全开放的. 本书的内容很紧凑，认真的读者最终会看到书中各部分之间的联系，以及我们选择这些材料的理由. 发现本书在体系结构方面的原理应该是读者的一个目标.

我们假定读者已经具有完成各种基本代数操作的能力.[1] 还假定读者熟悉逻辑和集合论的基本术语，如普通本科生在离散数学课程中学习的内容. 附录 A 总结了书中使用的各种记法. 如果在一些特定的算法里需要某些抽象代数的概念，书中会给出相应的定义. 我们还假定读者熟悉程序设计，理解计算机

1. 有关基本代数知识，推荐 Chrystal [1904].

体系结构,[2] 理解最基本的算法和数据结构.[3]

我们选用 C++, 是因为它组合了强有力的抽象设施和基础机器的正确表示.[4] 这里只用了该语言的一个小子集, 需求被写成程序里的结构化注释. 我们希望不熟悉 C++ 的读者也能阅读本书. 附录 B 描述了书中使用的 C++ 子集.[5] 在书中的任何地方, 在需要区分数学记法和 C++ 的地方, 根据所用的字体、排版和上下文就能确定用的是哪种意义 (是数学的还是 C++ 的). 虽然书中的许多概念和程序与 STL (C++ 标准模板库) 里的东西对应, 但这里的一些设计决策是与 STL 不同的. 书中还忽略了实际程序库 (如 STL) 必须考虑和处理的许多问题, 如名字空间、可见性、inline 指令等等.

第 1 章描述值、对象、类型、过程和概念. 第 2 ～ 5 章描述各种代数结构 (如半群、全序集) 上的算法. 第 6 ～ 11 章讨论抽象内存上的算法. 第 12 章讨论包含对象成员的对象. 跋给出了我们对本书中阐释的工作途径的反思.

致谢

真诚感谢 Adobe Systems 及其支持程序设计基础课程和本书的管理部门, 本书就是在这一课程中成长起来的. 特别感谢 Greg Gilley 启动了这个课程并建议我们写这本书; Dave Story 以及后来 Bill Hensler 提供了始终不渝的支持. 最后, 如果没有 Sean Parent 富于启发的管理以及对代码和正文持续的彻查, 本书也不可能完成. 本书的思想萌芽于我们与 Dave Musser 跨越了近三十年的紧密合作. 感谢 Bjarne Stroustrup 认真地发展 C++ 来支持这些想法. Dave 和 Bjarne 友善地专门到 San Jose 来并仔细审阅了早期的书稿. Sean Parent 和 Bjarne Stroustrup 为本书写了定义 C++ 子集的附录. Jon Brandt 审阅了本书的多个稿本. John Wilkinson 仔细阅读了最后的书稿, 提出了不计其数有价值的建议.

我们的编辑 Peter Gordon, 项目编辑 Elizabeth Ryan, 文本编辑 Evelyn Pyle, 审阅编辑 Matt Austern、Andrew Koenig、David Musser、Arch Robison、Jerry

2. 推荐 Patterson and Hennessy [2007].

3. 推荐 Tarjan [1983], 这是一本有关算法和数据结构的有选择的但又非常深刻的入门书.

4. 标准的参考书是 Stroustrup [2000].

5. 书中代码都能在 Microsoft Visual C++ 9 和 g++ 4 下编译运行. 所有代码, 以及几个使它们能编译直至做单元测试的简单的宏, 都可以在 www.elementsofprogramming.com 下载.

Schwarz、Jeremy Siek 和 John Wilkinson 都对本书做出了重要的贡献.

我们感谢在 Adobe 以及更早在 SGI 参与有关课程的学生们的各种建议. 我们希望自己成功地把这些课程里的材料编织为一个统一的整体. 我们衷心感谢 Dave Abrahams、Andrei Alexandrescu、Konstantine Arkoudas、John Banning、Hans Boehm、Angelo Borsotti、Jim Dehnert、John DeTreville、Boris Fomitchev、Kevlin Henney、Jussi Ketonen、Karl Malbrain、Mat Marcus、Larry Masinter、Dave Parent、Dmitry Polukhin、Jon Reid、Mark Ruzon、Geoff Scott、David Simons、Anna Stepanov、Tony Van Eerd、Walter Vannini、Tim Winkler 和 Oleg Zabluda 的意见和建议. 我们感谢 John Banning, Bob English、Steven Gratton、Max Hailperin、Eugene Kirpichov、Alexei Nekrassov、Mark Ruzon 和 Hao Song 指出了第一次印刷里的一些错误; Foster Brereton、Gabriel Dos Reis、Ryan Ernst、Abraham Sebastian、Mike Spertus、Henning Thielemann 和 Carla Villoria Burgazzi 指出了第二次印刷里的错误; Shinji Dosaka、Ryan Ernst、Steven Gratton、Volker Lukas、Qiu Zongyan、Abraham Sebastian 和 Sean Silva 指出了第三次印刷里的错误.[6]

最后, 还要衷心感谢通过其写作或者个人联系给我们教益, 使我们加深了对程序设计的理解的所有人以及相关的机构.

6. 最新的勘误信息见 www.elementsofprogramming.com.

关于作者

Alexander Stepanov 于 1967 到 1972 年间在国立莫斯科大学学习数学, 从 1972 年开始在苏联, 1977 年移民后继续在美国从事程序设计工作. 他编写过操作系统、程序设计工具、编译器和各种程序库. 他在程序设计基础方面的工作先后得到 GE、Brooklyn Polytechnic、AT&T、HP、SGI 和 Adobe 的支持. 他在 1995 年因 C++ 标准模板库的设计获 Dr. Dobb's Journal 的程序设计杰出贡献奖.

Paul McJones 于 1967 到 1971 年间在加州大学伯克利分校学习工程数学, 1967 年进入程序设计领域. 他涉足的领域包括操作系统、程序设计环境、事务处理系统, 以及企业和客户应用系统等. 他先后在加州大学、IBM、Xerox、Tandem、DEC 和 Adobe 工作. 1982 年他与合作者一起因论文 The Recovery Manager of the System R Database Manager 获得 ACM 程序设计系统和语言论文奖.

目　录

第 1 章

基础

本章从相关思想的一个简洁分类开始介绍一些术语：值、对象、类型、过程和概念，它们表示了与计算机有关的许多不同的理念范畴. 这里还要详细讨论本书的中心观点：规范性. 对一个过程，规范性意味着它对相等的参数总返回相等的结果. 对一个类型，规范性意味着它应该有相等运算符，以及保证相等关系的拷贝构造函数和赋值. 规范性使我们能应用 等值推理 (使用等值替换) 去变换和优化程序.

1.1 理念范畴：实体，类别，类属

为了解释什么是对象、类型，以及其他基本的计算机概念，概述一下与这些概念对应的理念范畴是很有帮助的.

抽象实体 (abstract entity) 指永存的不变的事物，而具体实体 (concrete entity) 指具体的个别的事物，其出现和存在与时间和空间有关. 一个属性 (attribute) 是具体实体与抽象实体之间的一种对应关系，它描述了该具体实体的某种性质、度量或者品质. 标识 (identity) 是我们感知实在世界的一种基本概念，它确定一个事物在随着时间变化中的不变性. 一个具体实体的属性可以改变，但这种改变不会影响其标识. 一个具体实体的一个快照 (snapshot) 就是在某个特定时间点上这一事物的所有属性的完整集合. 具体实体不仅包括所有物理上存在的实体，还包括法律的、经济的或者政治的实体. 蓝色和 13 是抽象实体的例子. 苏格拉底和美利坚合众国是具体实体的例子. 苏格拉底的眼睛的颜色和美国的州的个数是属性的例子.

一个抽象类别 (abstract species) 描述一批本质上等价的抽象实体的共性.

抽象类别的例子如自然数和颜色. 一个*具体类别* (concrete species) 描述一集本质上等价的具体实体的共性. 具体类别的例子如男人和美国的州.

一个*函数* (function) 是一套规则, 它将一个或几个取自某个或某些相应类别的抽象实体 (称为其*参量*, argument), 关联到来自某个抽象类别的一个抽象实体 (称为其*结果*, result). 函数的例子如后继函数, 它将每个自然数关联于紧随其后的那个自然数; 再如将两种颜色关联于它们的混合色的函数.

一个*抽象类属* (abstract genus) 描述在某些方面类似的一些不同的抽象类别. 抽象类属的例子如数和二元运算符. 一个*具体类属* (concrete genus) 描述在某些方面类似的一些不同的具体实体. 具体类属的例子如哺乳动物和鸟.

一个实体属于某个特定的类别, 这个类别确定了该实体的构造和存在的规则. 一个实体可以属于多个类属, 每个类属描述该实体的一些特定性质.

在本章的下面部分, 我们将论证对象和值都是实体, 类型是类别, 而概念是类属.

1.2 值

如果不知道如何解释, 在计算机里能看到的也就是一些 0 和 1. 一项*数据* (datum) 就是一个 0 和 1 的有穷序列.

一个*值类型* (value type) 是一个 (抽象或具体) 类别和一集数据之间的一个对应关系. 对应于某特定实体的数据称为该实体的一个*表示* (representation); 而这一实体称为相应数据的*解释* (interpretation). 我们把一项数据和它的解释称为一个*值* (value). 值的例子如采用大尾格式的 32 位二进制补码表示的整数; 或者用连续的两个 32 位二进制序列表示的有理数, 这两个二进制序列分别被解释为用整数表示的分子和分母, 而这两个整数又用大尾格式的 32 位二进制补码表示.

一项数据相对于一个值类型是*良形式的* (well-formed), 当且仅当这一数据表示了该类型里的一个抽象实体. 例如, 每个 32 位的二进制序列解释为模二补码的整数时都是良形式的; 而 IEEE 754 浮点的一个 NaN (Not a Number, 不是数) 解释为实数就不是良形式的.

一个值类型是*真部分的*(properly partial), 如果它的值只能表示对应抽象类别里的所有抽象实体的一个真子集; 否则它就是*全的* (total). 举例说, 类型 `int`

是真部分的, 而类型 bool 是全的.

一个值类型是*唯一表示的* (uniquely represented), 当且仅当每个抽象实体至多有一个对应的值. 例如, 如果表示真值的类型使用了一个字节, 其中的 0 解释为假, 而其他情况都解释为真, 那么这个类型就不是唯一表示的. 如果一个类型把整数表示为一个符号位和一个无符号量, 它就不能为 0 提供唯一表示. 用模二补码表示整数的类型是唯一表示的.

一个值类型是*有歧义的* (ambiguous), 当且仅在当该类型里存在具有多种解释的值. 有歧义的否定是*无歧义* (unambiguous). 例如, 将长于一个世纪的纪年表示为两个十进制数的类型就是有歧义的.

一个值类型里的两个值*相等* (equality), 当且仅当它们表示的是同一个抽象实体. 说两个值是*表示相等的* (representational equality), 当且仅当它们的数据是两个相同的 0/1 序列.

引理 1.1 如果一个值类型具有唯一表示, 相等就蕴涵着表示相等.

引理 1.2 如果一个值类型无歧义, 表示相等就蕴涵着相等.

如果一个值类型是唯一表示的, 通过检查其 0/1 序列是否相同的方式就可以实现相等判断. 否则就必须以一种与该类型的解释协调的方式来实现相等判断. 如果对于一个值类型经常需要生成新值而较少检查相等, 那就可以考虑选用非唯一性的表示, 因为这样做有可能使新值的生成更快, 而使相等判断较慢. 这类的例子如, 用一对整数表示的两个有理数相等, 条件是它们可以约简到同一个最简分数. 再如用非排序序列表示的两个集合相等, 条件是经过排序并消除重复元素后, 它们的对应元素相等.

有些时候, 实现真正的*行为上的* (behavioral) 相等判断的代价过大甚至根本就不可能. 例如, 将可计算函数编码为一个类型时的情况就是这样. 在这种情况下, 我们只能接受弱得多的*表示*相等的概念, 简单判断两个值的 0/1 序列是否相等.

计算机把抽象实体上的函数*实现* (implement) 为值上的*函数* (function). 虽然这些值驻留在内存, 但一个完好实现的值上的函数并不依赖于特定的内存地址, 它实现的是一个从值到值的映射 (mapping).

在一个值类型上定义的一个函数是*规范的* (regular), 当且仅当它遵从相等性: 给它的某个参数换一个相等的值, 它一定还得出相等的结果. 大多数的数

值函数都是规范的, 非规范的数值函数的一个例子是取分母函数: 假设有理数简单地表示为一对整数, 而该函数简单返回表示分子的那个整数. 例如 $\frac{1}{2} = \frac{2}{4}$, 但 numerator($\frac{1}{2}$) $\neq$ numerator($\frac{2}{4}$). 规范函数使我们可以做*等式推理* (equational reasoning), 允许我们做等值替换.

非规范的函数依赖于其参数的具体表示, 而不仅仅是参数的解释. 在设计一个值类型的表示时, 有两项相关的工作必须处理: 实现相等判断, 确定哪些函数应该是规范的.

1.3 对象

一个*存储* (memory) 就是一集存储字 (word), 其中每个字有一个*地址* (address) 和一项*内容* (content). 地址是一种固定大小的值, 这个大小称为*地址长度*. 而内容是另一固定大小的值, 其长度称为*字长*. 通过*装载* (load) 操作可以取得一个地址的内容. 通过*保存* (store) 操作改变一个地址所关联的内容. 存储的实例如计算机主存里的一组字节, 或磁盘驱动器里的一组区块.

一个*对象* (object) 就是一个具体实体的表示, 并且是作为某个存储里一个值. 对象有*状态* (state), 其状态就是某个值类型的一个值. 对象的状态可以改变. 对于给定的与某个具体实体对应的一个对象, 其状态对应于该实体的一个快照. 一个对象拥有一集*资源* (resource), 用于保存其状态, 如一些存储字或者文件里的一些记录.

即使某对象的值是一个连续的 0/1 序列, 保存这些 0/1 的资源也可以不是连续的. 相应的解释能给出这个对象的整体. 看一个例子: 两个 `double` 可以解释为一个复数, 为此并不要求它们紧邻存放. 一个对象的资源也完全可以位于不同的存储里. 但是本书只处理位于一个具有统一地址空间的存储里的对象, 这里的每个对象有一个唯一的*起始地址* (starting address), 从它出发可以找到该对象的所有资源.

一个*对象类型* (object type) 是一种在存储中保存和修改值的模式. 与每个对象类型相对应的有一个描述该类型对象的状态的值类型. 每个对象属于某一个对象类型. 作为对象类型的例子, 请考虑按 32 位模二补码方式, 采用小尾格式和 4 字节边界对齐表示的整数.

值和对象扮演着互补的角色. 值不会改变, 与在计算机里的特定实现方式

无关. 对象可以改变, 具有与具体计算相关的实现方式. 在任何一个时间点, 一个对象的状态都可以描述为一个值. 原则上说, 这个值可以写在纸上 (做出一个快照) 或者序列化 (serialize) 后通过通信链路传输. 用值的方式描述对象的状态, 在讨论相等性时就可以抽象掉对象的具体实现方式. 函数式程序设计处理的是值; 而命令式程序设计处理的是对象.

我们用值来表示实体. 因为值是不变的, 它们可以表示抽象实体. 也可以用值的序列来表示某具体实体的一些快照的序列. 对象保存着代表实体的值. 由于对象可以变化, 因此可以用于表示具体实体, 并通过令其不断取得新值的方式表示该实体的变化. 也可用对象表示抽象实体, 这时它们将保持不变, 或者取相应抽象实体的一些不同近似值.

在计算机里使用对象, 有如下三个原因.

1. 对象能模拟可以改变的具体实体, 例如工资系统里的雇员记录.

2. 对象为实现值上的函数提供了强有力的支持, 例如写一个用迭代算法实现浮点数平方根的过程.

3. 带存储的计算机是目前唯一可用的通用计算设备.

值类型的某些性质也可用于对象类型. 一个对象是*良形式的* (well-formed), 当且仅当其状态是良形式的. 一个对象类型是*真部分的* (properly partial), 当且仅当其值类型是真部分的; 否则它就是*全的* (total). 一个对象类型是*唯一表示的* (uniquely represented), 当且仅当其值类型是唯一表示的.

由于具体实体有标识, 表示它们的对象也要有与之对应的概念. 一个*标识符号* (identity token) 是一个唯一值, 它表示相应对象的标识, 可以从相应对象的值及其资源的地址计算出来. 标识符号的例子如对象的地址, 或者到保存着该对象的数组里的下标, 或者人事记录里的雇员编号. 对标识符号的相等检查对应于实体标识的相等检查. 在应用系统运行期间, 一个特定对象有可能在一个数据结构里移动, 也可能从一个数据结构移动到另一个, 这种移动有可能改变对象的标识符号.

同样类型的两个对象*相等* (equality), 当且仅当它们的状态相等. 如果两个对象相等, 我们就说其中的一个是另一个的*拷贝* (copy). 修改一个对象并不会对它的任何拷贝产生影响.

本书将使用一种编程语言, 在该语言里不能脱离对象和对象类型去描述值和值类型. 由于这种情况, 从现在开始, 只要提到类型时没加修饰词, 那么就是指对象类型.

1.4 过程

一个过程 (procedure) 是一个指令序列, 它修改某些对象的状态, 也可能构造或者销毁一些对象.

根据程序员的意图, 与一个过程交互的对象分为四类.

1. 输入输出 (input/output), 指该过程通过其参数或返回值直接或间接传递的那些对象.

2. 局部状态 (local state), 该过程在其一次调用期间创建、销毁或修改的那些对象.

3. 全局状态 (global state), 该过程和其他过程在多次调用中访问的对象.

4. 拥有的状态 (own state), 仅供该过程 (及其隶属过程) 访问但又能被该过程的多次调用共享的对象.

如果一个对象作为参数或结果传递, 称它是直接传递的; 如果是通过指针或类似指针的机制传递, 则称它是间接传递的. 如果一个对象被某个过程读但是并不修改, 称它是该过程的输入. 如果一个对象被某过程写入、创建或销毁, 但该过程并不访问其初始状态, 则称它是该过程的输出. 如果一个对象被某过程既读又修改, 称它是该过程的输入/输出.

一个类型的计算基 (computational basis) 是有穷的一集过程, 基于它们可以构造出该类型上的其他过程. 一组基是高效的 (efficient), 当且仅当基于它实现的任何过程的效率不比基于任何其他基写出的过程低效. 例如, 对 k-位无符号整数的一组只提供 0, 相等判断和后继操作的基就不是高效的, 因为基于后继函数实现加法的复杂性是 k 的指数函数.

一组基是有表达力的 (expressive), 当且仅当基于它可以紧凑而方便地定义相关类型上的其他过程. 特别是, 如果合适的话, 一个基应该提供所有常用的数学运算. 举例说, 虽然减法运算可以通过求负和加法实现, 但在一个有表达力

的基中应该直接提供它. 与此类似, 求负也可以通过减法和 0 实现, 但在有表达力的基中也应直接提供.

1.5 规范类型

存在这样一组过程, 如果把它们包含到一个类型的计算基中, 就能方便地把对象放入各种数据结构, 或者通过算法把对象从一个数据结构复制到另一数据结构. 我们称具有这样的基的类型为*规范的* (regular), 因为使用这样的类型可以保证程序行为的规范性, 进而获得类型之间的互操作性.[1] 可以从内部类型, 如 `bool`、`int`, 以及限制到良形式值的 `double`, 看到规范类型的语义. 一个类型是*规范的*, 当且仅当它的基包含了相等检查、赋值、析构操作、默认构造操作、拷贝构造操作、一个全序判断[2] 和一个基础类型.[3]

相等判断是一个过程, 它以同类型的两个对象为参数, 当且仅当两个对象的状态相等时返回真. 同样应该定义不等判断, 它应返回相等判断的否定. 我们使用下面记法:

	规程	C++
相等	a = b	a == b
不等	a ≠ b	a != b

赋值 (assignment) 是一个过程, 它以同类型的两个对象为参数, 使得第一个对象等于第二个, 但并不修改第二个对象. 赋值的意义不依赖于第一个对象的初值. 我们使用下面记法:

	规程	C++
赋值	a ← b	a = b

析构操作 (destructor) 是一个过程, 它结束一个对象的存在. 对一个对象调用析构操作之后, 就不能再将任何过程作用于它, 而且它以前的存储位置和资源都可以用于其他用途了. 析构操作经常被隐式地调用. 全局对象在应用程序

1. 虽然规范性是 STL 的设计基础, 其正式定义最早出现在文献 Dehnert and Stepanov [2000] 中.
2. 严格的说法要到第 4 章才能说清楚, 它可以是一个全序, 或者一个默认的全序.
3. 基础类型在第 12 章定义.

终止时销毁 (析构), 局部对象在它们声明所在的块退出时销毁, 数据结构的元素在数据结构销毁时也被销毁.

构造操作 (constructor) 是一个过程, 它把一些存储位置变换到一个对象. 其可能行为可以是什么也不做, 也可以是创建极其复杂的对象.

一个对象处于部分成形 (partially formed) 状态, 如果它已经可以赋值或销毁. 对于部分成形但尚未完全成形的对象, 除了赋值 (放在左边) 和析构, 做其他任何过程的效果都无定义.

引理 1.3 良形式的对象也是部分成形的.

默认构造操作 (default constructor) 没有参数, 且能使对象达到部分成形的状态. 我们将采用下面记法:

	C++
类型 T 的局部对象	`T a;`
类型 T 的匿名对象	`T()`

拷贝构造操作 (copy constructor) 有一个同类型的参数, 它构造出一个等于该参数的新对象. 我们将采用下面记法:

	C++
对象 b 的局部拷贝	`T a = b;`

1.6 规范过程

一个过程是*规范的*, 当且仅当将其输入替换为任何相等的对象, 它给出的结果与以前相等. 就像值类型一样, 在定义一个对象类型时, 必须在如何实现类型里的相等, 以及哪些过程应该具有规范性方面有一种统一的考虑.

练习 1.1 请将规范的概念扩展到过程的输入输出对象 (即那些既读又修改的对象).

虽然规范性应该是最基本的选择, 也有些因素要求有非规范过程.

1. 返回一个对象的地址的过程; 例如, 内部函数 `addressof`.
2. 返回由真实世界的状态确定值的过程, 如返回时钟或其他设备的值.

3. 返回依赖于自己拥有的状态的值的过程; 例如伪随机数生成器.

4. 返回与一个对象的表示相关的某些属性的值的过程, 例如为数据结构保留的存储量.

函数式过程 (functional procedure) 是在规范类型上定义的一类规范过程, 它们有一个或几个直接输入和一个作为过程结果的返回值. 函数式过程的规范性使我们可以采用两种技术为其传递输入. 如果参数的规模较小, 或者过程里需要参数的可修改拷贝, 那么可以用*值*的方式传递它, 做出参数的一个局部拷贝. 否则可以用*常量引用*的方式传递它. 函数式过程可以实现为 C++ 的函数、函数指针或者函数对象.[4]

下面是一个函数式过程:

```
int plus_0(int a, int b)
{
    return a + b;
}
```

下面是一个语义等价的函数式过程:

```
int plus_1(const int& a, const int& b)
{
    return a + b;
}
```

下面过程的语义也与上面两个等价, 但它不是函数式的, 其输入和输出参数都是间接传递的:

```
void plus_2(int* a, int* b, int* c)
{
    *c = *a + *b;
}
```

在 plus_2 里, a 和 b 是输入对象, c 是输出对象. 函数式过程的概念是语法性质而不是语义性质: 按我们的术语, plus_2 是规范的但不是函数式的.

4. C++ 的函数不是对象, 不能作为参数传递; C++ 函数指针和函数对象是对象, 可以作为参数传递.

一个函数式过程的*定义空间* (definition space) 是其预设的可能输入值集合的一个子集. 函数式过程对于其定义空间里的输入总终止; 而对超出其定义空间的输入, 它可能不终止, 也或许不能返回有意义的值.

同源的 (homogeneous) 函数式过程的输入对象都属于同一个类型. 同源函数式过程的*定义域* (domain) 是其输入的类型. 我们并不把非同源的函数式过程的定义域说成是其输入类型的直积, 而是个别地讨论过程的各个输入类型.

一个函数式过程的*值域* (codomain) 是其输出的类型. 函数式过程的*结果空间* (result space) 是其值域的一个子集, 是该过程从其定义空间取得输入后返回的所有可能值的集合.

考虑函数式过程

```
int square(int n) { return n * n; }
```

其定义域和值域都是 int, 其定义空间是平方值在此类型里可以表示的所有整数的集合, 而其结果空间是这个类型里可以表示的平方整数的集合.

练习 1.2 假设 int 是 32-位的模二补码类型, 请精确给出上面函数式过程的定义空间和结果空间.

1.7 概念

如果一个过程使用了一个类型, 它就会依赖于该类型的语法、语义, 还有其计算基的复杂性. 在语法上, 它依赖于一些确定的文字量和一些具有特定名字和*签名* (signature) 的过程的存在; 其语义依赖于基过程的语义; 其复杂性依赖于基过程的时间和空间复杂性. 如果用另一个具有同样性质的类型取代这个类型, 程序将仍然是正确的. 如果不是基于具体的类型, 而是基于对类型的一些要求 (通过语法和语义性质描述) 来设计软件部件, 例如设计库过程或数据结构, 一定能提高它们的可用性. 我们将这样的一组要求称为一个*概念* (concept). 类型表示类别; 而概念表示类属.

要描述概念, 就需要有一些处理类型的机制, 包括类型属性、类型函数和类型构造符. *类型属性* (type attribute) 是从一个类型到一个值的映射, 它描述有关类型的某种特征. 类型属性的例子如 C++ 里提供的内部类型属性

sizeof(T), 一个类型的对象的对齐方式, 以及一个 struct 的成员个数等. 如果 F 是一个函数式过程类型, Arity(F) 返回其输入的个数.

类型函数 (type function) 是从一个类型到一个从属类型的映射. 类型函数的一个例子是: 从给定的 "到 T 的指针" 得到类型 T. 在某些情况下, 定义一个带有一个附加的整数参数的*带索引* (indexed) 类型函数可能很有用. 例如定义一个类型函数, 它返回一个结构类型的第 i 个成员的类型 (从 0 开始计). 如果 F 是一个函数式过程类型, 类型函数 Codomain(F) 返回其结果的类型. 如果 F 是一个函数式过程且 $i < \mathrm{Arity}(F)$, 索引类型函数 InputType(F, i) 返回其第 i 个参数的类型 (从 0 开始计).[5]

类型构造符 (type constructor) 是从一些已有类型出发构造新类型的机制. 举例说, pointer(T) 是一个内部提供的类型构造符, 它从一个类型 T 出发返回 "指向 T 的指针" 类型; struct 是一种内部提供的 n 元类型构造符; 一个结构模板是一个用户定义的 n 元类型构造符.

如果 $\mathcal{T}$ 是一个 n 元类型构造符, 下面用 $\mathcal{T}_{\mathsf{T}_0,\ldots,\mathsf{T}_{n-1}}$ 表示将它应用于类型 $\mathsf{T}_0,\ldots,\mathsf{T}_{n-1}$. 一个重要例子是 pair (二元组), 将其应用于规范类型 T_0 和 T_1, 就得到了一个 struct 类型 $\mathsf{pair}_{\mathsf{T}_0,\mathsf{T}_1}$, 它有一个类型为 T_0 的成员 m0 和一个类型为 T_1 的成员 m1. 要保证类型 $\mathsf{pair}_{\mathsf{T}_0,\mathsf{T}_1}$ 本身也是规范的, 就要求它的相等、赋值、析构和构造操作都是基于类型 T_0 和 T_1, 通过按两个成员分别做的方式定义. 同样的技术可以用于任何其他元组类型, 例如 triple (三元组). 第 12 章将说明如何实现 $\mathsf{pair}_{\mathsf{T}_0,\mathsf{T}_1}$, 并说明更复杂的类型构造符如何维持规范性.

采用更形式化一点的说法, 一个*概念* (concept) 是对一个或多个类型的需求的一个描述, 这一描述以对类型上的过程、类型属性和类型函数的存在及其相关性质的方式给出. 如果某个 (某些) 特定类型满足这些需求, 就说这一概念被这个 (或这些) 类型*建模* (modeled), 或者说它们是这一概念的*模型* (model). 要断言概念 $\mathcal{C}$ 被类型 $\mathsf{T}_0,\ldots,\mathsf{T}_{n-1}$ 建模, 我们写 $\mathcal{C}(\mathsf{T}_0,\ldots,\mathsf{T}_{n-1})$. 如果任意的满足概念 $\mathcal{C}'$ 的类型也必定满足概念 $\mathcal{C}$, 就说概念 $\mathcal{C}'$ *精化* (refine) 概念 $\mathcal{C}$. 如果 $\mathcal{C}'$ 精化 $\mathcal{C}$, 也说 $\mathcal{C}$ *弱于* $\mathcal{C}'$.

类型概念 (type concept) 指定义在一个类型上的一个概念. 举例说, C++ 定义了类型概念*整数类型*, 它被*无符号整数类型*和*有符号整数类型*精化. 而 STL 定义了类型概念*序列* (sequence). 前面一直用基本类型概念 *Regular* 和

5. 附录 B 说明了怎样在 C++ 里定义类型属性和类型函数.

FunctionalProcedure, 它们对应于前面给出的非形式化定义.

我们可以用标准的数学记法来形式化地定义各种概念. 要定义概念 $\mathfrak{C}$, 采用的写法是

$$\begin{aligned}&\mathfrak{C}(\mathsf{T}_0,\ldots,\mathsf{T}_{n-1}) \triangleq\\ &\quad \mathcal{E}_0\\ &\wedge\ \mathcal{E}_1\\ &\wedge\ \ldots\\ &\wedge\ \mathcal{E}_{k-1}\end{aligned}$$

其中的 $\triangleq$ 读作 "按定义相等", T_i 是形式类型参数, $\mathcal{E}_j$ 是概念子句. 概念子句可以有如下三种形式:

1. 前面已定义的概念的应用, 说明相应类型参数的一个子集建模此概念.
2. 类型属性、类型函数或者过程签名, 建模相应概念的类型里必须有它们. 过程签名的形式是 $\mathsf{f}:\mathsf{T}\to\mathsf{T}'$, 其中 T 是过程的定义域而 T' 是其值域. 类型函数签名的形式是 $\mathsf{F}:\mathfrak{C}\to\mathfrak{C}'$, 其定义域和值域都是概念.
3. 基于这些类型属性、类型函数和过程表述的公理.

我们有时也在第二类概念子句里的类型属性、类型函数或过程的签名之后包含它们的定义. 这种定义的形式是 $\mathsf{x}\mapsto\mathcal{F}(\mathsf{x})$, 其中的 $\mathcal{F}$ 是表达式. 在特定模型里, 这一定义可以被另一个不同的但与之协调的实现所覆盖.

举个例子, 下面概念描述的是一元函数式过程:

$$\begin{aligned}&\mathit{UnaryFunction}(\mathsf{F}) \triangleq\\ &\quad \mathit{FunctionalProcedure}(\mathsf{F})\\ &\wedge\ \mathsf{Arity}(\mathsf{F}) = 1\\ &\wedge\ \mathsf{Domain} : \mathit{UnaryFunction} \to \mathit{Regular}\\ &\quad \mathsf{F} \mapsto \mathsf{InputType}(\mathsf{F}, 0)\end{aligned}$$

下面概念描述的是同源的函数式过程:

$$\begin{aligned}&\mathit{HomogeneousFunction}(\mathsf{F}) \triangleq\\ &\quad \mathit{FunctionalProcedure}(\mathsf{F})\end{aligned}$$

$$\wedge\ \mathsf{Arity}(\mathsf{F}) > 0$$
$$\wedge\ (\forall \mathsf{i}, \mathsf{j} \in \mathbb{N})(\mathsf{i}, \mathsf{j} < \mathsf{Arity}(\mathsf{F})) \Rightarrow (\mathsf{InputType}(\mathsf{F}, \mathsf{i}) = \mathsf{InputType}(\mathsf{F}, \mathsf{j}))$$
$$\wedge\ \mathsf{Domain} : \mathit{HomogeneousFunction} \rightarrow \mathit{Regular}$$
$$\mathsf{F} \mapsto \mathsf{InputType}(\mathsf{F}, 0)$$

应该看到

$$(\forall \mathsf{F} \in \mathit{FunctionalProcedure})\ \mathit{UnaryFunction}(\mathsf{F}) \Rightarrow \mathit{HomogeneousFunction}(\mathsf{F})$$

抽象 (abstract) 过程指用类型和常量参数化的过程, 带有对这些参数的要求.[6] 下面将为此使用函数模板和函数对象模板. 参数放在 `template` 关键字后面, 类型参数用 `typename` 引入, 常量值用 `int` 或其他整数类型名引入. 对函数的需求通过 `requires` 子句严格描述, 该子句的内容是一个表达式, 基于常量值、具体类型、形式参数、类型属性和类型参数的应用, 值和类型的相等, 相关概念, 以及逻辑连接词等描述.[7]

这里是一个抽象过程的例子:

```
template<typename Op>
    requires(BinaryOperation(Op))
Domain(Op) square(const Domain(Op)& x, Op op)
{
    return op(x, x);
}
```

定义域里的值可能很大, 所以这里通过常量引用的方式传递该参数. 操作一般说比较小 (例如是函数指针或很小的函数对象), 因此用值方式传递.

概念描述的是一个类型的所有对象都满足的性质, 其中的**前条件** (precondition) 描述特定对象的性质. 举例说, 一个过程可能要求它的某个参数是素数, 对整数类型的这一需求就需要用一个概念描述, 其中的素数性质用一个前条件描述. 函数指针的类型只描述了它的签名, 未描述其语义性质. 例如, 一个过

---

6. 本质上具有我们所采用的形式的抽象过程出现在 1930 van der Waerden [1930], 基于 Emmy Noether 和 Emil Artin 的演讲, George Collins 和 David Musser 20 世纪 60 年代后期和 70 年代前期的论文在计算机代数的研究中使用了它们. 例如可以参考 Musser [1975].

7. `requires` 子句的完整语法见附录 B.

程可能要求它的一个参数是指向函数的指针, 并要求函数实现的是整数上的一个可结合的二元运算. 有关整数上二元运算的要求可以用一个概念描述, 而函数的结合性用一个前条件描述.

为一集类型定义一个前条件, 需要使用一些数学记法, 例如全称和存在量词, 蕴涵等. 举例说, 要描述整数的素数性质, 用下面定义

property(N : *Integer*)

prime : N

$$n \mapsto (|n| \neq 1) \wedge (\forall u, v \in N)\, uv = n \Rightarrow (|u| = 1 \vee |v| = 1)$$

这里的第一行引入了一些形式类型参数和它们建模的概念, 第二行说明一个性质并给出其签名, 第三行是描述有关的参数应满足的特定性质的谓词.

一元函数式过程的规范性可以定义如下

property(F : *UnaryFunction*)

regular_unary_function : F

$$f \mapsto (\forall f' \in F)(\forall x, x' \in \mathrm{Domain}(F))$$
$$(f = f' \wedge x = x') \Rightarrow (f(x) = f'(x'))$$

这一定义很容易扩充到 n 元函数: 将相等的函数作用于相等的参数得到相等的结果. 通过扩充, 我们要求规范的抽象函数的所有实例化也都是规范的. 除非另有说明, 本书后面说到过程时都是指规范过程, 因此下面讨论中将不明确写出这一前条件.

项目 1.1 请将相等、赋值、赋值构造操作扩充到不同类型的对象上. 请考虑两个类型的解释和联系起跨类型的过程的公理.

1.8 总结

人们公认的对于现实世界的许多常识在计算机里都有所表现. 通过将值和对象的意义落实到它们的解释, 可以得到一种简单而具有内在一致性的看法. 如果将对应的实体纳入我们的考虑, 各种设计决策 (例如如何定义相等) 就会变得直接而清晰.

第 2 章

变换及其轨道

本章把变换定义为从一个类型到它自身的规范函数. 从一个初始值开始连续应用一个变换, 就得到了这个值的轨道. 我们要实现一个能确定轨道结构的算法, 它只依赖于变换的规范性和轨道的有穷性. 这个算法可以应用到不同的领域. 例如, 可以用它检查一个链接表是否为循环的, 或用于分析一个伪随机数生成器. 我们将为这一算法推导出一套接口, 形式上是一集过程及其参数和结果的定义. 对轨道结构算法的分析, 使我们可以用最简单的方式介绍自己的编程方法.

2.1 变换

虽然从任意一组类型到任意类型的函数都存在, 但是有些具有特殊签名的函数类更为常用. 本书中经常用的是两类函数: 同源谓词和同源运算. 同源谓词在形式上都是 $\mathsf{T} \times \cdots \times \mathsf{T} \to \mathsf{bool}$; 同源运算都是形如 $\mathsf{T} \times \cdots \times \mathsf{T} \to \mathsf{T}$ 的函数. 一般而言存在着任意的 n 元谓词和 n 元运算, 但实际中遇到最多的还是一元和二元的同源谓词, 一元和二元的同源运算.

谓词是返回真值的函数:

$$
\begin{aligned}
&\mathit{Predicate}(\mathsf{P}) \triangleq \\
&\qquad \mathit{FunctionalProcedure}(\mathsf{P}) \\
&\quad \wedge\ \mathsf{Codomain}(\mathsf{P}) = \mathsf{bool}
\end{aligned}
$$

同源谓词也是同源函数:

$$
\mathit{HomogeneousPredicate}(\mathsf{P}) \triangleq
$$

$$
\begin{aligned}
&\mathit{Predicate}(\mathsf{P}) \\
\wedge\ &\mathit{HomogeneousFunction}(\mathsf{P})
\end{aligned}
$$

一元谓词只有一个参数:

$$
\begin{aligned}
&\mathit{UnaryPredicate}(\mathsf{P}) \triangleq \\
&\quad \mathit{Predicate}(\mathsf{P}) \\
&\wedge\ \mathit{UnaryFunction}(\mathsf{P})
\end{aligned}
$$

运算是同源函数, 其值域与作用域相同:

$$
\begin{aligned}
&\mathit{Operation}(\mathsf{Op}) \triangleq \\
&\quad \mathit{HomogeneousFunction}(\mathsf{Op}) \\
&\wedge\ \mathsf{Codomain}(\mathsf{Op}) = \mathsf{Domain}(\mathsf{Op})
\end{aligned}
$$

下面是几个同源运算的实例:

```
int abs(int x) {
    if (x < 0) return -x; else return x;
} // 一元运算

double euclidean_norm(double x, double y) {
    return sqrt(x * x + y * y);
} // 二元运算

double euclidean_norm(double x, double y, double z) {
    return sqrt(x * x + y * y + z * z);
} // 三元运算
```

引理 2.1 $\mathsf{euclidean\_norm}(x, y, z) = \mathsf{euclidean\_norm}(\mathsf{euclidean\_norm}(x, y), z)$

这一引理说明上面的三元运算可以从相应的二元版本得到. 但是, 由于效率, 表达方便, 或者准确性等原因, 这样的三元运算也可以纳入处理三维空间的程序的计算基.

如果一个过程的定义空间是其输入类型的直积的子集, 就称该过程为部分的 (partial); 如果其定义空间等于其输入类型的直积, 则说它是全的. 按标准的

数学习惯, 我们也认为部分过程包括全过程, 并称那些不是全的部分过程为非全的 (nontotal). 由于计算机表示的有穷性, 有些全函数在计算机里的实现却是非全的. 例如, 有符号的 32 位整数加法就是非全的.

一个非全过程需要有一个描述其定义空间的前条件. 要验证对这个过程的一个调用是正确的, 必须确定所给的实参满足这一前条件. 有时我们会把部分过程作为参数传给一个算法, 因此需要在运行时确定这种过程参数的定义空间问题. 为了处理这类情况, 我们将定义一个**定义空间谓词** (definition space predicate), 它与这一过程参数的输入相同, 当且仅当实际输入在这个过程的定义空间里时, 让这个谓词返回真. 在调用一个非全过程之前, 或者其前条件必须满足, 或者该调用是用这个过程的定义空间谓词保护的.

练习 2.1 请为 32 位有符号整数的加法实现一个定义空间谓词.

本章研究一元运算, 我们称其为 **变换** (transformation):

$$
\begin{aligned}
&\mathit{Transformation}(\mathsf{F}) \triangleq \\
&\quad \mathit{Operation}(\mathsf{F}) \\
&\quad \wedge\ \mathit{UnaryFunction}(\mathsf{F}) \\
&\quad \wedge\ \mathsf{DistanceType} : \mathit{Transformation} \to \mathit{Integer}
\end{aligned}
$$

这里的 DistanceType 将在下一节讨论.

变换可以**自组合** (self composable), 例如可以写 f(x), f(f(x)), f(f(f(x))), 等等. f(f(x)) 的定义空间是 f 的定义空间和结果空间的交. 这种自组合能力和检查相等能力的结合, 使我们可以定义许多有趣的算法.

设 f 是一个变换, 它的幂 (power) 定义如下:

$$
\mathsf{f}^n(x) = \begin{cases} x & \text{if } n = 0, \\ \mathsf{f}^{n-1}(\mathsf{f}(x)) & \text{if } n > 0 \end{cases}
$$

要实现计算 $\mathsf{f}^n(x)$ 的算法, 就要描述对一个整数类型的需求. 第 5 章将研究描述与整数有关的一些概念, 现在我们暂时依靠对整数的一些直观理解. 有符号和无符号的整数类型都是整数的模型, 还有任意精度的整数等, 它们都包含下面的运算和文字量:

	规程	C++
加	+	+
减	−	-
乘	·	*
除	/	/
取模	mod	%
零	0	I(0)
一	1	I(1)
二	2	I(2)

其中的 I 是一个整数类型.

这样就可以写出下面算法:

```
template<typename F, typename N>
    requires(Transformation(F) && Integer(N))
Domain(F) power_unary(Domain(F) x, N n, F f)
{
    // 前条件: n ⩾ 0 ∧ (∀i ∈ N) 0 < i ⩽ n ⇒ f^i(x) 有定义
    while (n != N(0)) {
        n = n - N(1);
        x = f(x);
    }
    return x;
}
```

2.2 轨道

要理解一个变换的全局行为, 可以考察其**轨道** (orbit) 的结构. 所谓变换的轨道, 就是从一个开始元素出发, 通过反复应用这一变换能到达的所有元素. 如果对某个 $n \geqslant 0$ 有 $y = f^n(x)$, 就说 y 是从 x 出发在变换 f 下**可达的** (reachable). 如果有某个 $n \geqslant 1$ 使 $x = f^n(x)$, 就说 x 在 f 下是**环路的** (cyclic). 如果 x 不在 f 的定义空间里, 称 x 在 f 下是**终止点** (terminal). x 在变换 f 下的**轨道** (orbit) 就是 x

在 f 下可达的所有元素的集合.

引理 2.2 一条轨道不会同时包含环路元素和终止元素.

引理 2.3 一条轨道至多包含一个终止元素.

如果元素 y 为从 x 在 f 下可达, 从 x 到 y 的距离 (distance) 定义为从 x 得到 y 的最小的变换步数. 显然, 距离并不总有定义.

给定变换类型 F, 类型 DistanceType(F) 是一个足够大的整数类型, 足以表示任何变换 $f \in F$ 从 $T = \mathsf{Domain}(F)$ 的一个元素到另一元素的最大变换步数. 如果类型 T 用 k 个二进制位表示, 它就有 2^k 个值, 而不同值之间只有 $2^k - 1$ 个变换步. 这样, 如果 T 是固定大小的类型, 同样大小的整数类型就能作为表示 T 上任意变换的距离的合法类型. (可以不用距离类型, 而是在 power_unary 里用任意整数类型, 因为这种额外推广不会在这里造成问题.) 一种常见情况是同一定义域上的变换都具有同样的距离类型. 这时类型函数 DistanceType 就对这一定义域类型有定义, 定义了与相关变换类型对应的类型函数.

如果类型函数 DistanceType 存在, 就可以写出下面过程:

```
template<typename F>
    requires(Transformation(F))
DistanceType(F) distance(Domain(F) x, Domain(F) y, F f)
{
    // 前条件: y 在 f 下从 x 可达
    typedef DistanceType(F) N;
    N n(0);
    while (x != y) {
        x = f(x);
        n = n + N(1);
    }
    return n;
}
```

轨道有不同的形状. x 在某变换下的轨道可以是

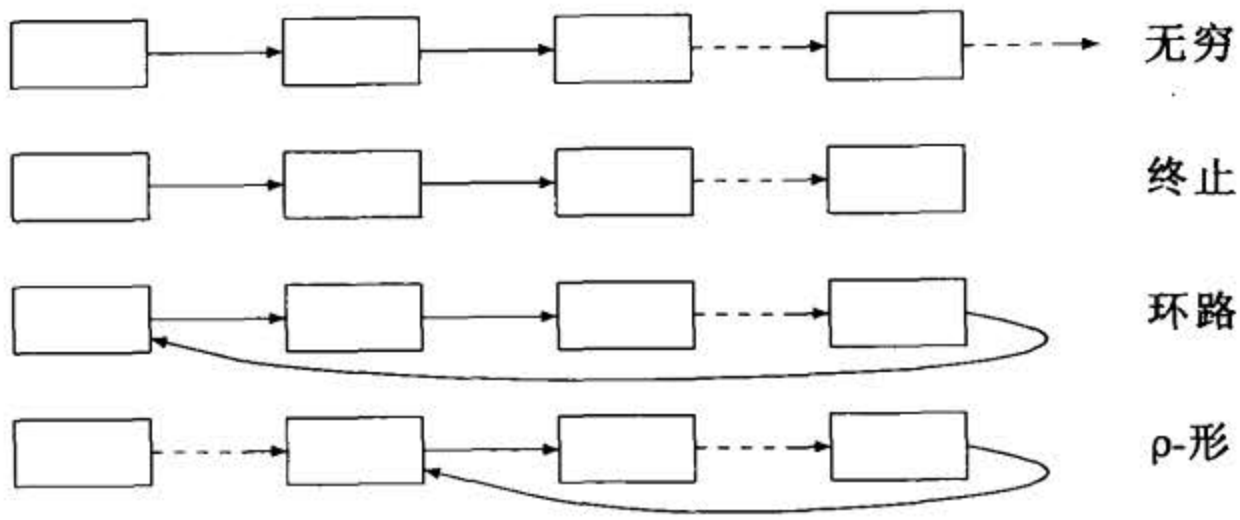


图 2.1 轨道的形状

无穷的　若它既不是终止的也没有环路

终止的　若它有一个终止元素

环路的　若 x 是环路的

ρ 形的　若 x 本身不是环路的, 但它的轨道包含环路元素

如果 x 的轨道不是无穷的, 它就是有穷的. 图 2.1 给出了轨道的各种情况.

轨道环 (cycle of orbit) 是轨道的一集元素, 规定轨道为无穷或终止时其环为空. 轨道柄 (handle of orbit) 是轨道里除去环之外的部分, 环路轨道的柄为空. 连接点 (connection point of orbit) 是轨道上的第一个环路元素, 对环路轨道取其第一个元素, 对 ρ 形轨道是轨道柄之后的第一个元素. 轨道规模 (size of orbit) o 是轨道中不同元素的个数; 轨道的柄规模 (handle size) h 是轨道柄中元素的个数; 环规模 (circle size) c 是轨道环中元素的个数.

引理 2.4 $o = h + c$

引理 2.5 从轨道中任意一点到该轨道的环路中任意一点的距离都有定义.

引理 2.6 若 x 和 y 是规模为 c 的环里的两个不同点, 那么

$$c = \mathsf{distance}(x, y, f) + \mathsf{distance}(y, x, f)$$

引理 2.7 若 x 和 y 是规模为 c 的环的两个不同点, x 到 y 的距离满足

$$0 \leqslant \mathsf{distance}(x, y, f) < c$$

2.3 碰撞点

如果不知道定义而只能观察其行为, 我们无法确定一个变换的一条特定轨道是否无穷, 因为它完全可能在任意一点结束或者循环. 如果知道一条轨道是有穷的, 那就可以用一个算法来确定该轨道的形状了. 因此, 本章针对轨道的所有算法都有一个有关轨道有穷性的隐含前条件.

显然可以写一个简单而平凡的算法, 让它保存已访问的每个元素, 在每一步检查新遇到的元素是否曾经访问过. 虽然可以用哈希技术来加快搜索, 但这个算法至少需要线性的存储空间, 而这种要求对于许多实际应用是不能接受的. 还好, 存在着只需要常量存储空间的算法.

下面的类比有助于理解这一算法. 如果有一辆速度快的汽车和一辆速度慢的汽车在一条道路上行驶, 当且仅当存在环路时快车将能追上慢车. 如果没有环路, 快车将比慢车先到达道路的终点. 如果有环路, 在慢车进入环路之后, 快车一定会在环路中追上它. 下面我们把这种直观想法从连续域转到离散域, 还要小心地避免快车超过慢车.[1]

这一算法的离散版本基于找到快车遭遇慢车的点. 变换 f 和起始点 x 确定的碰撞点 (collision point) 就是那个唯一的 y, 它使

$$y = f^n(x) = f^{2n+1}(x)$$

其中的 $n \geqslant 0$ 是满足这一条件的最小整数. 这一定义给出了一个用于确定轨道结构的算法, 它只需比较较快的迭代和较慢的迭代. 如果要处理的是部分变换, 那就必须给算法送一个定义空间谓词:

```
template<typename F, typename P>
    requires(Transformation(F) && UnaryPredicate(P) &&
        Domain(F) == Domain(P))
Domain(F) collision_point(const Domain(F)& x, F f, P p)
{
    // 前条件: p(x) ⇔ f(x) 有定义
    if (!p(x)) return x;
    Domain(F) slow = x;          // slow = f^0(x)
```

1. Knuth [1997, 7 页] 将这一算法归功于 Robert W. Floyd.

```
    Domain(F) fast = f(x);          // fast = f^1(x)
                                    // n ← 0 (完成迭代)
    while (fast != slow) {          // slow = f^n(x) ∧ fast = f^(2n+1)(x)
        slow = f(slow);             // slow = f^(n+1)(x) ∧ fast = f^(2n+1)(x)
        if (!p(fast)) return fast;
        fast = f(fast);             // slow = f^(n+1)(x) ∧ fast = f^(2n+2)(x)
        if (!p(fast)) return fast;
        fast = f(fast);             // slow = f^(n+1)(x) ∧ fast = f^(2n+3)(x)
                                    // n ← n + 1
    }
    return fast;                    // slow = f^n(x) ∧ fast = f^(2n+1)(x)
    // 后条件: 返回终止点或碰撞点的值
}
```

我们将通过三个步骤来确立 collision_point 算法的正确性: (1) 验证 f 不会被应用于定义空间之外的参数; (2) 验证如果它终止, 后条件一定满足; 以及 (3) 验证它必定终止.

即使 f 是部分函数, 它在这一过程中的使用也总是良定义的, 因为 fast 的移动受到 p 调用的保护. 而 slow 的移动不需要保护, 这是由于 f 的规范性: slow 和 fast 在同一轨道上, 所以将 f 应用于 slow 时总有定义.

算法里的注释说明, 如果经过 $n \geqslant 0$ 次迭代后 fast 变得等于 slow, 那就一定有 $\text{fast} = f^{2n+1}(x)$ 和 $\text{slow} = f^n(x)$. 进一步说, n 就是满足这一条件的最小整数, 因为对每个 $i < n$ 都已经检查过这个条件.

如果没有环路, 由于有穷性, p 将最终返回假. 如果有环路, slow 最终将到达连接点 (环路的第一个点). 在程序里的循环语句开始处考虑当 slow 刚进入环路时从 fast 到 slow 的距离 d, 有 $0 \leqslant d < c$. 如果 $d = 0$ 过程立即结束. 否则从 fast 到 slow 的距离在每次迭代中减 1. 这就说明该过程总会终止, 而当其终止时 slow 移动了总共 $h + d$ 步.

下面过程确定一条轨道是否终止:

```
template<typename F, typename P>
    requires(Transformation(F) && UnaryPredicate(P) &&
```

```
        Domain(F) == Domain(P))
bool terminating(const Domain(F)& x, F f, P p)
{
    // 前条件: p(x) ⇔ f(x) 有定义
    return !p(collision_point(x, f, p));
}
```

有时我们知道一个变换或者是全的, 或者从某个特定开始点出发的轨道是不终止的. 对于这种情况, 下面特殊版本的 collision_point 会很有用:

```
template<typename F>
    requires(Transformation(F))
Domain(F)
collision_point_nonterminating_orbit(const Domain(F)& x, F f)
{
    Domain(F) slow = x;          // slow = f^0(x)
    Domain(F) fast = f(x);       // fast = f^1(x)
                                 // n ← 0 (完成迭代)
    while (fast != slow) {       // slow = f^n(x) ∧ fast = f^(2n+1)(x)
        slow = f(slow);          // slow = f^(n+1)(x) ∧ fast = f^(2n+1)(x)
        fast = f(fast);          // slow = f^(n+1)(x) ∧ fast = f^(2n+2)(x)
        fast = f(fast);          // slow = f^(n+1)(x) ∧ fast = f^(2n+3)(x)
                                 // n ← n + 1
    }
    return fast;                 // slow = f^n(x) ∧ fast = f^(2n+1)(x)
    // 后条件: 返回碰撞点的值
}
```

要想确定环路的结构, 即确定轨道的柄规模、连接点和环规模, 就需要分析碰撞点的位置.

当过程返回碰撞点时

$$f^n(x) = f^{2n+1}(x)$$

其中 n 是 slow 走过的步数, $2n+1$ 是 fast 走过的步数. 而且

$$n = h + d$$

这里的 h 是柄规模, $0 \leqslant d < c$ 是 slow 在环里走过的步数. fast 走过的步数是

$$2n + 1 = h + d + qc$$

这里的 $q > 0$ 是 fast 碰到 slow 时完成整个环的次数. 由于 $n = h + d$, 所以

$$2(h + d) + 1 = h + d + qc$$

通过化简得到

$$qc = h + d + 1$$

用下式表示 h 对 c 取模:

$$h = mc + r$$

其中 $0 \leqslant r < c$. 代换给出

$$qc = mc + r + d + 1$$

也就是

$$d = (q - m)c - r - 1$$

$0 \leqslant d < c$ 意味着

$$q - m = 1$$

所以

$$d = c - r - 1$$

而且需要 $r + 1$ 步去完成整个环.

这样, 从碰撞点到连接点的距离就是

$$e = r + 1$$

对环路轨道有 $h = 0$ 且 $r = 0$, 从碰撞点到轨道开始点的距离是

$$e = 1$$

由此可知, 环路性质可以用下面过程检查:

```
template<typename F>
    requires(Transformation(F))
bool circular_nonterminating_orbit(const Domain(F)& x, F f)
{
    return x == f(collision_point_nonterminating_orbit(x, f));
}

template<typename F, typename P>
    requires(Transformation(F) && UnaryPredicate(P) &&
        Domain(F) == Domain(P))
bool circular(const Domain(F)& x, F f, P p)
{
    // 前条件: p(x) ⇔ f(x) 有定义
    Domain(F) y = collision_point(x, f, p);
    return p(y) && x == f(y);
}
```

至此仍不可能知道柄规模 h 和环规模 c. 在知道碰撞点之后很容易确定后者, 为此只需遍历环路一次并记录步数.

要想确定 h, 先看看碰撞点的位置:

$$f^{h+d}(x) = f^{h+c-r-1}(x) = f^{mc+r+c-r-1}(x) = f^{(m+1)c-1}(x)$$

从冲突点走 $h+1$ 步就到达了点 $f^{(m+1)c+h}(x)$, 它等于 $f^h(x)$, 因为 $(m+1)c$ 对应于绕环 $m+1$ 次. 如果同时从 x 走 h 步并从碰撞点走 $h+1$ 步, 两者就会在连接点相遇. 换句话说, x 的轨道和过碰撞点 1 步的点在恰好 h 步后汇合, 这一情况揭示了下面几个算法:

```
template<typename F>
    requires(Transformation(F))
Domain(F) convergent_point(Domain(F) x0, Domain(F) x1, F f)
{
    // 前条件: (∃n ∈ DistanceType(F)) n ⩾ 0 ∧ f^n(x0) = f^n(x1)
    while (x0 != x1) {
```

```
        x0 = f(x0);
        x1 = f(x1);
    }
    return x0;
}

template<typename F>
    requires(Transformation(F))
Domain(F)
connection_point_nonterminating_orbit(const Domain(F)& x, F f)
{
    return convergent_point(
        x,
        f(collision_point_nonterminating_orbit(x, f)),
        f);
}

template<typename F, typename P>
    requires(Transformation(F) && UnaryPredicate(P) &&
        Domain(F) == Domain(P))
Domain(F) connection_point(const Domain(F)& x, F f, P p)
{
    // 前条件: p(x) ⇔ f(x) 有定义
    Domain(F) y = collision_point(x, f, p);
    if (!p(y)) return y;
    return convergent_point(x, f(y), f);
}
```

引理 2.8 如果两个元素的轨道相交, 它们将有同样的环路元素.

练习 2.2 请设计一个算法, 对于给定的变换和定义空间谓词, 它能确定两个元素的轨道是否相交.

练习 2.3 convergent_point 的前条件保证它终止. 请针对没有该条件, 但已知在两点 x0 和 x1 的轨道上有共同元素的情况实现算法 convergent_point_guarded.

2.4 轨道规模的度量

对于类型 T 上的轨道, 有关规模 o, h 和 c 的自然类型应该是一个足以记录类型 T 中所有不同的值的个数的整数计数类型. 如果类型 T 占了 k 位, 它至多有 2^k 个值, 所以一个 k 位的计数类型将不能表示从 0 到 2^k 的所有计数值. 但在使用距离类型时有一种表示这些规模的方法.

一条轨道有可能包含某类型的所有值, 这时 o 就可能无法存入相应的距离类型. 对不同形状的轨道, h 或 c 也可能无法存入. 然而对 ρ-形轨道, h 和 c 一定能存入. 对所有情况下面几个量都可以存入: $o-1$ (轨道中的最大距离), $h-1$ (柄里的最大距离), 以及 $c-1$ (环里的最大距离). 这使我们可以实现一个过程, 它返回一个三元组来表示轨道的完整结构, 其三个成员分别是:

情况	m0	m1	m2
终止	$h-1$	0	终止元素
环路	0	$c-1$	x
ρ-形	h	$c-1$	连接点

```
template<typename F>
    requires(Transformation(F))
triple<DistanceType(F), DistanceType(F), Domain(F)>
orbit_structure_nonterminating_orbit(const Domain(F)& x, F f)
{
    typedef DistanceType(F) N;
    Domain(F) y = connection_point_nonterminating_orbit(x, f);
    return triple<N, N, Domain(F)>(distance(x, y, f),
                                   distance(f(y), y, f),
                                   y);
}
```

```
template<typename F, typename P>
    requires(Transformation(F) &&
        UnaryPredicate(P) && Domain(F) == Domain(P))
triple<DistanceType(F), DistanceType(F), Domain(F)>
orbit_structure(const Domain(F)& x, F f, P p)
{
    // 前条件: p(x) ⇔ f(x) 有定义
    typedef DistanceType(F) N;
    Domain(F) y = connection_point(x, f, p);
    N m = distance(x, y, f);
    N n(0);
    if (p(y)) n = distance(f(y), y, f);
    // 终止时: m = h − 1 ∧ n = 0
    // 否则: m = h ∧ n = c − 1
    return triple<N, N, Domain(F)>(m, n, y);
}
```

练习 2.4 请推导出本章各算法中使用不同操作 (f, p, 相等) 的次数的公式.

练习 2.5 用 orbit_structure_nonterminating_orbit 确定在你所使用的平台上的伪随机数生成器对不同种子值的平均柄规模和环路规模.

2.5 动作

在算法里, 变换 f 经常被用在如下形式的语句中

```
x = f(x);
```

应用变换去改变对象的状态, 就定义了相应对象上的一个动作 (action). 变换和动作之间有直接的对偶关系, 可以基于变换来定义动作, 反之亦然:

```
void a(T& x) { x = f(x); }    // 从变换定义动作
```

以及

```
T f(T x) { a(x); return x; } // 从动作定义变换
```

虽然存在这种对偶关系, 但有时独立的实现效率更高, 这种情况下就应该分别提供动作和变换. 举例说, 如果在很大的对象上定义变换, 但是只修改整个状态中很小的一部分, 采用动作定义就可能大大提高效率.

练习 2.6 请用动作的方式重写本章的各个算法.

项目 2.1 检查环路的另一种方式是反复检查一个不断更新的元素是否与另一个保存着的元素相等, 与此同时按一定间隔更新那个保存的元素. Sedgewick et al. [1982], Brent [1980] 和 Levy [1982] 讨论了这种想法和其他想法. 请为轨道分析写一些算法, 针对不同应用比较它们的性能, 并为选择适当的算法提出一组建议.

2.6 总结

抽象使我们可以定义出一些能用于不同领域的抽象过程. 规范性和函数的类型在这里起着最重要的作用: fast 和 slow 能走同一条轨道的基础就是规范性. 开发一批专有术语 (例如, 轨道的类型和规模), 是最基础最重要的工作. 一些附属类型, 例如距离类型, 也需要精确地定义.

第 3 章

可结合运算

本章讨论可结合的二元运算. 可结合性使人可以重组相邻的运算, 基于这种重组能力可以得到一个计算二元运算的幂的有效算法. 规范性使我们可以用许多程序变换来优化这一算法. 我们随后要利用该算法在对数时间里计算各种线性递归, 例如计算斐波那契数.

3.1 可结合性

二元运算是有两个参数的运算:

$$\begin{aligned} &\mathit{BinaryOperation}(\mathrm{Op}) \triangleq \\ &\quad \mathit{Operation}(\mathrm{Op}) \\ &\quad \wedge\ \mathsf{Arity}(\mathrm{Op}) = 2 \end{aligned}$$

二元的加法和乘法是数学的核心概念, 常用的这类东西很多, 如求较小或较大的值, 合取和析取, 集合的交与并等. 这些运算都是可结合的 (associative):

$$\begin{aligned} &\textbf{property}(\mathrm{Op} : \mathit{BinaryOperation}) \\ &\mathsf{associative} : \mathrm{Op} \\ &\quad \mathrm{op} \mapsto (\forall a, b, c \in \mathsf{Domain}(\mathrm{op}))\, \mathrm{op}(\mathrm{op}(a, b), c) = \mathrm{op}(a, \mathrm{op}(b, c)) \end{aligned}$$

当然, 也存在不可结合的二元运算, 例如除法和减法.

如果根据上下文完全可以看清所用的可结合二元运算 op, 我们常用隐式的乘法记法将其写成 ab 而不是 $\mathrm{op}(a, b)$. 由于可结合性, 在涉及两次或多次应用 op 的表达式里不需要括号, 因为所有分组方式都等价: $(\cdots(a_0a_1)\cdots)a_{n-1} = \cdots = a_0(\cdots(a_{n-2}a_{n-1})\cdots) = a_0a_1\cdots a_{n-1}$. 当 $a_0 = a_1 = \cdots = a_{n-1} = a$ 时将其写成 a^n, 称为 a 的 n 次幂 (power).

引理 3.1 $a^n a^m = a^m a^n = a^{n+m}$ (同一元素的幂可交换.)

引理 3.2 $(a^n)^m = a^{nm}$

当然, $(ab)^n = a^n b^n$ 未必总成立, 只在运算还可以交换时成立.

如果 f 和 g 是同一定义域上的变换, 它们的复合 (composition) $g \circ f$ 也是一个变换, 将 x 映射到 $g(f(x))$.

引理 3.3 二元运算的复合是可结合的.

在可结合运算 op 的定义域中选一个元素 a, 并把表达式 $op(a, x)$ 看作只有一个形参 x 的一元运算, 就可以把 a 看作是一个 "乘以 a" 运算. 这说明可以采用与变换的幂 f^n 的同样记法, 将一个元素在可结合二元运算下的幂记为 a^n 是合适的. 这一对偶性质使我们可以用取自前一章的一个算法, 证明一个有关可结合运算的幂的有趣定理. 如果对某个可结合运算, 存在整数 $0 < n < m$ 使 $x^n = x^m$, 则称 x 在该运算下具有*有穷的阶* (finite order). 如果对某可结合运算有 $x = x^2$, 则称 x 是该运算的*幂等元素* (idempotent element).

定理 3.1 一个有穷阶元素必定有一个幂等的幂 (Frobenius [1895]).

证明. 设 x 是可结合运算 op 下的一个有穷阶元素. 令 $g(z) = op(x, z)$. 由于 x 的阶有穷, 它在 g 下的轨道有循环. 根据定理的条件, 存在 $n \geqslant 0$ 使得

$$\mathsf{collision\_point}(x, g) = g^n(x) = g^{2n+1}(x)$$

这样

$$g^n(x) = x^{n+1}$$
$$g^{2n+1}(x) = x^{2n+2} = x^{2(n+1)} = (x^{n+1})^2$$

而且 x^{n+1} 就是 x 的幂等的幂. □

引理 3.4 也可以在上面的证明里用 collision_point_nonterminating_orbit.

3.2 计算乘幂

对可结合运算 op 计算 a^n 的算法以 a, n 和 op 为参数, 这里 a 的类型是 op 的定

义域, n 必须属于某个整数类型. 如果没有可结合性假设, 可以用两个算法分别完成从左到右或从右到左的计算:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_left_associated(Domain(Op) a, I n, Op op)
{
    // 前条件: n > 0
    if (n == I(1)) return a;
    return op(power_left_associated(a, n - I(1), op), a);
}

template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_right_associated(Domain(Op) a, I n, Op op)
{
    // 前条件: n > 0
    if (n == I(1)) return a;
    return op(a, power_right_associated(a, n - I(1), op));
}
```

这两个算法都应用相应的运算 $n-1$ 次. 对于非可结合的运算, 它们可能返回不同结果. 作为例子, 请考虑减法运算的 1 到 3 次幂的结果.

当 a 和 n 都是整数而运算是乘法时, 这两个算法给出的都是乘幂; 运算是加法时两个算法做的都是乘法. 古埃及人早已发现了更快的乘法算法, 可以将其推广到计算任意可结合运算的幂.[1]

可结合性使人可以自由地将运算分组, 由此可得

$$a^n = \begin{cases} a & \text{若 } n = 1 \\ (a^2)^{n/2} & \text{若 } n \text{ 是偶数} \\ (a^2)^{\lfloor n/2 \rfloor} a & \text{若 } n \text{ 是奇数} \end{cases}$$

1. 源自 Robins and Shute [1987, 16–17 页]; 莎草纸古写本的大致年代是公元前 1650 年, 但写本上的注记说明它是大约公元前 1850 年的另一古写本的抄本.

这对应于

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_0(Domain(Op) a, I n, Op op)
{
    // 前条件: associative(op) ∧ n > 0
    if (n == I(1)) return a;
    if (n % I(2) == I(0))
        return power_0(op(a, a), n / I(2), op);
    return op(power_0(op(a, a), n / I(2), op), a);
}
```

现在对指数 n 算一下 power_0 执行运算的次数. 递归调用的次数是 $\lfloor \log_2 n \rfloor$. 令 ν 为 n 的二进制表示里 1 的个数. 每次递归调用执行一次运算求 a 的平方, 还需要 $\nu - 1$ 次额外的运算调用, 所以总运算次数是

$$\lfloor \log_2 n \rfloor + (\nu - 1) \leqslant 2\lfloor \log_2 n \rfloor$$

对于 $n = 15$, $\lfloor \log_2 n \rfloor = 3$, 其表示中有 4 个 1, 公式给出的是 6 次运算. 另一不同分组方式是 $a^{15} = (a^3)^5$, 这里 a^3 做 2 次运算 而 a^5 做 3 次, 总数为 5. 对另外一些指数也存在更快的分组方式, 例如对 23, 27, 39 和 43.[2]

power_left_associated 需要做 $n - 1$ 次运算, 而 power_0 最多做 $2\lfloor \log_2 n \rfloor$ 次运算. 很明显, 对于较大的 n, power_0 总是快得多. 但事情也不都这样. 例如, 要是做具有任意精度整数系数的一般多项式的乘法, power_left_associated 会更快些.[3] 还有, 即使对这样的简单算法, 我们也不知道如何精确描述一种复杂性需求, 使之可用于确定两者中哪个更好.

power_0 能处理很大的指数, 如 10^{300}, 这使它在密码学中非常重要.[4]

---

2. 有关最小运算数求幂的深入讨论见 Knuth [1997, 465–481 页].

3. 见 McCarthy [1986].

4. 见 Rivest et al. [1978] 中有关 RSA 的工作.

3.3 程序变换

power_0 是有关算法的一个令人满意的实现, 它适用于运算的代价高于函数递归调用开销的情况. 本节要推导出一个迭代算法, 它执行运算的次数和 power_0 一样. 这里将要做一系列程序变换, 这些变换也可以用在其他许多情况中.[5] 在本书后面的部分, 通常将只给出算法的最终版本或几乎最终版本.

power_0 包含两个相同的递归调用, 它每次只执行其中一个. 这使我们可能通过*公共子表达式删除*技术来缩小代码的规模:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_1(Domain(Op) a, I n, Op op)
{
    // 前条件: associative(op) ∧ n > 0
    if (n == I(1)) return a;
    Domain(Op) r = power_1(op(a, a), n / I(2), op);
    if (n % I(2) != I(0)) r = op(r, a);
    return r;
}
```

现在的目标是删除递归调用, 为此要做的第一步是把过程变换到*尾递归形式* (tail-recursive form), 其中在过程执行的最后一步是对自身的递归调用. 完成该变换的一种技术是*引入累积变量* (accumulation-variable introduction), 用于在不同递归调用之间携带累积的结果:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate_0(Domain(Op) r, Domain(Op) a, I n,
                              Op op)
{
    // 前条件: associative(op) ∧ n ⩾ 0
    if (n == I(0)) return r;
```

5. 只有在运算的语义和复杂性已知的情况下, 编译器才会对一些内部类型做类似变换. 规范性概念是类型创建者的一个断言, 它保证程序员和编译器可以安全地执行这些变换.

```
    if (n % I(2) != I(0)) r = op(r, a);
    return power_accumulate_0(r, op(a, a), n / I(2), op);
}
```

设 r_0, a_0 和 n_0 是 r, a 和 n 的原值, 下面不变式 (recursion invariant) 在每次递归调用时都成立: $ra^n = r_0 a_0^{n_0}$. 这个版本还有另一优点, 它不仅计算幂, 还能计算乘以一个系数的幂. 它也处理了指数为 0 的情况. 但是在指数从 1 变到 0 时 power_accumulate_0 将多做一次平方. 增加一种情况就可以消除它:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate_1(Domain(Op) r, Domain(Op) a, I n,
                              Op op)
{
    // 前条件: associative(op) ∧ n ≥ 0
    if (n == I(0)) return r;
    if (n == I(1)) return op(r, a);
    if (n % I(2) != I(0)) r = op(r, a);
    return power_accumulate_1(r, op(a, a), n / I(2), op);
}
```

增加额外情况导致重复出现的子表达式, 也使三个检测不独立了. 通过仔细分析检测之间的依赖性和顺序, 考虑它们的出现频率, 可以给出

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate_2(Domain(Op) r, Domain(Op) a, I n,
                              Op op)
{
    // 前条件: associative(op) ∧ n ≥ 0
    if (n % I(2) != I(0)) {
        r = op(r, a);
        if (n == I(1)) return r;
    } else if (n == I(0)) return r;
```

```
    return power_accumulate_2(r, op(a, a), n / I(2), op);
}
```

在一个尾递归过程里, 如果所有递归调用中的过程形参都是对应的实参, 它就是一个*严格尾递归的* (strict tail-recursive) 过程:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate_3(Domain(Op) r, Domain(Op) a, I n,
                              Op op)
{
    // 前条件: associative(op) ∧ n ⩾ 0
    if (n % I(2) != I(0)) {
        r = op(r, a);
        if (n == I(1)) return r;
    } else if (n == I(0)) return r;
    a = op(a, a);
    n = n / I(2);
    return power_accumulate_3(r, a, n, op);
}
```

严格尾递归过程可以变换为一个迭代过程, 方法是把每个递归调用代换为一个到过程开始的 goto, 也可以用一个等价的迭代结构:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate_4(Domain(Op) r, Domain(Op) a, I n,
                              Op op)
{
    // 前条件: associative(op) ∧ n ⩾ 0
    while (true) {
        if (n % I(2) != I(0)) {
            r = op(r, a);
            if (n == I(1)) return r;
```

```
        } else if (n == I(0)) return r;
        a = op(a, a);
        n = n / I(2);
    }
}
```

递归不变式变成了这里的循环不变式 (loop invariant).

如果开始时 $n > 0$, 在变成 0 前要先经过 1. 我们借用这种情况消去对 0 的检查并加强前条件 (strengthening precondition):

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate_positive_0(Domain(Op) r,
                                       Domain(Op) a, I n,
                                       Op op)
{
    // 前条件: associative(op) ∧ n > 0
    while (true) {
        if (n % I(2) != I(0)) {
            r = op(r, a);
            if (n == I(1)) return r;
        }
        a = op(a, a);
        n = n / I(2);
    }
}
```

知道了 $n > 0$ 会很有用. 在开发组件的过程中经常会发现新的接口情况.

现在放松前条件 (relaxing precondition):

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate_5(Domain(Op) r, Domain(Op) a, I n,
                              Op op)
```

```
{
    // 前条件: associative(op) ∧ n ⩾ 0
    if (n == I(0)) return r;
    return power_accumulate_positive_0(r, a, n, op);
}
```

通过一个简单的等式, 就可以用power_accumulate 实现 power:

$$a^n = aa^{n-1}$$

这一变换就是消去累积变量 (accumulation-variable elimination):

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_2(Domain(Op) a, I n, Op op)
{
    // 前条件: associative(op) ∧ n > 0
    return power_accumulate_5(a, a, n - I(1), op);
}
```

这个算法多做了一些不必要的运算. 例如, 当 n 是 16 时它要执行 7 次运算, 其中只有 4 次是必要的. 当 n 是奇数时这个算法很好. 避免上述问题的方法是反复做 a 的平方, 并不断将指数折半直至它变成奇数:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_3(Domain(Op) a, I n, Op op)
{
    // 前条件: associative(op) ∧ n > 0
    while (n % I(2) == I(0)) {
        a = op(a, a);
        n = n / I(2);
    }
    n = n / I(2);
    if (n == I(0)) return a;
```

```
    return power_accumulate_positive_0(a, op(a, a), n, op);
}
```

练习 3.1 请自己确认最后三行代码是正确的.

3.4 处理特殊情况的过程

在上面的最后版本里用到下面运算:

```
n / I(2)
n % I(2) == I(0)
n % I(2) != I(0)
n == I(0)
n == I(1)
```

其中 / 和 % 的代价很高. 对无符号整数或有符号整数的非负值, 可以用移位和掩码运算来代替它们.

识别出程序里经常出现的表达式, 其中涉及一些过程或某种类型的常量. 将它们定义为相应的特殊情况过程常常很有价值. 针对特殊情况的实现经常比一般情况的处理更高效, 因此应该把这类过程放入类型的计算基. 对语言的内部类型, 通常存在针对特殊情况的机器指令. 对用户定义类型, 针对特殊情况的优化也常有显著效果. 举例说, 两个任意多项式的除法比一个多项式除以 x 难得多. 与此类似, 两个高斯整数 (形式为 a + bi 的数, 其中 a 和 b 都是整数而 $i = \sqrt{-1}$) 相除比一个高斯整数除以 1 + i 难得多.

任何整数类型都应该提供下面的特殊情况过程:

$$
\begin{aligned}
&\mathit{Integer}(\mathrm{I}) \triangleq \\
&\qquad \mathsf{successor} : \mathrm{I} \to \mathrm{I} \\
&\qquad\qquad \mathrm{n} \mapsto \mathrm{n} + 1 \\
&\quad \wedge\ \mathsf{predecessor} : \mathrm{I} \to \mathrm{I} \\
&\qquad\qquad \mathrm{n} \mapsto \mathrm{n} - 1 \\
&\quad \wedge\ \mathsf{twice} : \mathrm{I} \to \mathrm{I} \\
&\qquad\qquad \mathrm{n} \mapsto \mathrm{n} + \mathrm{n}
\end{aligned}
$$

∧ half_nonnegative : I → I

$n \mapsto \lfloor n/2 \rfloor$, 其中 $n \geqslant 0$

∧ binary_scale_down_nonnegative : I × I → I

$(n, k) \mapsto \lfloor n/2^k \rfloor$, 其中 $n, k \geqslant 0$

∧ binary_scale_up_nonnegative : I × I → I

$(n, k) \mapsto 2^k n$, 其中 $n, k \geqslant 0$

∧ positive : I → bool

$n \mapsto n > 0$

∧ negative : I → bool

$n \mapsto n < 0$

∧ zero : I → bool

$n \mapsto n = 0$

∧ one : I → bool

$n \mapsto n = 1$

∧ even : I → bool

$n \mapsto (n \bmod 2) = 0$

∧ odd : I → bool

$n \mapsto (n \bmod 2) \neq 0$

练习 3.2 请为 C++ 的各整数类型实现上面这些过程.

现在可以给出求幂过程的最后实现了, 其中用到一些特殊情况过程:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate_positive(Domain(Op) r,
                                     Domain(Op) a, I n,
                                     Op op)
{
    // 前条件: associative(op) ∧ positive(n)
    while (true) {
      if (odd(n)) {
          r = op(r, a);
```

```
            if (one(n)) return r;
        }
        a = op(a, a);
        n = half_nonnegative(n);
    }
}

template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate(Domain(Op) r, Domain(Op) a, I n,
                            Op op)
{
    // 前条件: associative(op) ∧ ¬negative(n)
    if (zero(n)) return r;
    return power_accumulate_positive(r, a, n, op);
}

template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power(Domain(Op) a, I n, Op op)
{
    // 前条件: associative(op) ∧ positive(n)
    while (even(n)) {
        a = op(a, a);
        n = half_nonnegative(n);
    }
    n = half_nonnegative(n);
    if (zero(n)) return a;
    return power_accumulate_positive(a, op(a, a), n, op);
}
```

由于已知 $a^{n+m} = a^n a^m$, 表达式 a^0 必须求出运算 op 的单位元. 可以把单

位元作为操作的另一参数传入, 将 power 的定义扩充到包括 0 次幂:[6]

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power(Domain(Op) a, I n, Op op, Domain(Op) id)
{
    // 前条件: associative(op) ∧ ¬negative(n)
    if (zero(n)) return id;
    return power(a, n, op);
}
```

项目 3.1 浮点乘法和加法不可结合, 用于作为 power 或 power_left_associated 的运算就可能得到不同结果. 请设法弄清, 在求浮点数的整数次幂时, 是 power 还是 power_left_associated 能给出更准确的结果.

3.5 参数化算法

power 在为算法提供运算时采用了两种不同技术:

1. 可结合运算通过参数传递. 这使 power 可以用于同一类型的不同运算, 例如用于模 n 的乘法.

2. 指数上的运算被作为指数类型的计算基的一部分. 例如, 这里采用为 power 传一个 half_nonnegative 参数的方式, 是因为不知道是否存在某种情况, 其中需要在同一类型上为 half_nonnegative 提供另一种实现方式.

一般而言, 当一个算法可以用于同样类型的不同运算时, 就应该考虑通过参数传递运算. 当过程以一个运算为参数时, 如果可能就应该提供一个合适的默认运算. 例如, 传给 power 的最自然的默认运算是乘法.

一个运算符或过程名表示了不同类型里语义相同的多个运算的情况称为*重载* (overloading), 此时也说该运算符或过程名是*在类型上重载的*. 典型的例子是 +, 它被用于自然数、有理数、多项式和矩阵等. 数学里总用 + 表示可结

6. 另一技术是定义一个函数 identity_element, 使 identity_element(op) 返回 op 的单位元.

合并可交换的运算, 因此用 + 表示字符串拼接就有些不合理. 与此类似, 如果同时有 + 和 ×, × 必须在 + 上分配. power 里的 half_nonnegative 也是在指数类型上重载的.

实例化一个抽象过程, 例如实例化 collision_point 或者 power, 建立的是一些重载过程. 如果实际的类型参数满足需求, 同一抽象过程的多个实例将具有相同的语义.

3.6 线性递归

k 阶的*线性递归函数* (linear recurrence function) 是一个函数 f, 对它有

$$f(y_0, \cdots, y_{k-1}) = \sum_{i=0}^{k-1} a_i y_i$$

其中系数 $a_0, a_{k-1} \neq 0$. 序列 $\{x_0, x_1, \cdots\}$ 是一个 k 阶的*线性递归序列*, 如果存在一个 k 阶线性递归函数, 例如 f, 使得

$$(\forall n \geqslant k)\, x_n = f(x_{n-1}, \ldots, x_{n-k})$$

请注意, 这里的下标 x 是递减的. 给定了 k 个*初始值* $x_0, \ldots, x_{k-1}$ 和一个 k 阶的线性递归函数, 我们可以通过一个明显的算法生成一个线性递归序列. 对所有 $n \geqslant k$ 计算 x_n, 该算法要求应用函数 $n-k+1$ 次. 下面将看到, 利用 power, 只需 $O(\log_2 n)$ 步就可以算出 x_n.[7] 如果 $f(y_0, \ldots, y_{k-1}) = \sum_{i=0}^{k-1} a_i y_i$ 是一个 k 阶线性递归函数, 可以认为 f 是执行了一个向量内积运算:[8]

$$\begin{bmatrix} a_0 & \cdots & a_{k-1} \end{bmatrix} \begin{bmatrix} y_0 \\ \vdots \\ y_{k-1} \end{bmatrix}$$

如果把系数向量扩充为从对角线为 1 的*伴随矩阵* (companion matrix), 就可以在计算 x_n 的新值的同时将相应的老值 $x_{n-1}, \ldots, x_{n-k+1}$ 移到下次迭代所

7. 线性递归的第一个 $O(\log n)$ 算法应归功于 Miller and Brown [1966].

8. 按线性代数的观点, 见 Kwak and Hong [2004]. 有关线性递归的讨论从 214 页开始.

需的正确位置:

$$\begin{bmatrix} a_0 & a_1 & a_2 & \cdots & a_{k-2} & a_{k-1} \\ 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 \end{bmatrix} \begin{bmatrix} x_{n-1} \\ x_{n-2} \\ x_{n-3} \\ \vdots \\ x_{n-k} \end{bmatrix} = \begin{bmatrix} x_n \\ x_{n-1} \\ x_{n-2} \\ \vdots \\ x_{n-k+1} \end{bmatrix}$$

由于矩阵乘法具有可结合性, 易知, 将 k 个初始值的向量乘以该伴随矩阵的 $n-k+1$ 次幂, 就能得到 x_n:

$$\begin{bmatrix} x_n \\ x_{n-1} \\ x_{n-2} \\ \vdots \\ x_{n-k+1} \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 & \cdots & a_{k-2} & a_{k-1} \\ 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 \end{bmatrix}^{n-k+1} \begin{bmatrix} x_{k-1} \\ x_{k-2} \\ x_{k-3} \\ \vdots \\ x_0 \end{bmatrix}$$

利用 power, 通过至多 $2\log_2(n-k+1)$ 次矩阵乘法就可以得到 x_n. 而直截了当的矩阵乘法需要做 k^3 次系数乘法和 k^3-k^2 次系数加法. 这样, x_n 的计算需要做不多于 $2k^3\log_2(n-k+1)$ 次系数乘法和 $2(k^3-k^2)\log_2(n-k+1)$ 次系数加法. 应记得, k 是线性递归的阶, 而且是一个常数.[9]

这里没有定义线性递归序列的元素类型. 它可以是整数、有理数、 实数或复数. 只要求它存在可结合并可交换的加法和乘法, 而且乘法对加法分配.[10]

由 2 阶线性递归函数

$$\mathsf{fib}(y_0, y_1) = y_0 + y_1$$

从初始值 $f_0 = 0$ 和 $f_1 = 1$ 生成的序列 f_i 称为斐波那契序列.[11] 用 power 和 2×2 矩阵乘法可以直接计算第 n 个斐波那契数 f_n. 下面用斐波那契序列计算说明, 对这一特殊实例, 说明其中的 k^3 次乘法能如何减少. 令

$$F = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

---

9. Fiduccia [1985] 说明了可以怎样通过模多项式乘法减小这里的常数因子.

10. 这就是说, 允许任何是半群的模型的类型. 半群概念将在第 5 章定义.

11. Leonardo Pisano, *Liber Abaci*, 第一版, 1202. 英译本见 Sigler [2002]. 该序列在 404 页.

为通过线性递归生成斐波那契序列的伴随矩阵. 可以证明

$$F^n = \begin{bmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{bmatrix}$$

确实如此:

$$F^1 = \begin{bmatrix} f_2 & f_1 \\ f_1 & f_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$\begin{aligned} F^{n+1} &= FF^n \\ &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{bmatrix} \\ &= \begin{bmatrix} f_{n+1} + f_n & f_n + f_{n-1} \\ f_{n+1} & f_n \end{bmatrix} = \begin{bmatrix} f_{n+2} & f_{n+1} \\ f_{n+1} & f_n \end{bmatrix} \end{aligned}$$

这使我们可以把 F^m 和 F^n 的矩阵乘积写成

$$\begin{aligned} F^m F^n &= \begin{bmatrix} f_{m+1} & f_m \\ f_m & f_{m-1} \end{bmatrix} \begin{bmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{bmatrix} \\ &= \begin{bmatrix} f_{m+1}f_{n+1} + f_m f_n & f_{m+1}f_n + f_m f_{n-1} \\ f_m f_{n+1} + f_{m-1} f_n & f_m f_n + f_{m-1} f_{n-1} \end{bmatrix} \end{aligned}$$

矩阵 F^n 可以基于其最下一行表示为一个 pair (f_n, f_{n-1}), 因为顶行可以用 $(f_{n-1} + f_n, f_n)$ 计算. 这样就得到了下面代码:

```
template<typename I>
    requires(Integer(I))
pair<I, I> fibonacci_matrix_multiply(const pair<I, I>& x,
                                     const pair<I, I>& y)
{
    return pair<I, I>(
        x.m0 * (y.m1 + y.m0) + x.m1 * y.m0,
        x.m0 * y.m0 + x.m1 * y.m1);
}
```

这一过程只需执行 4 次乘法, 而不像一般的 2×2 矩阵乘法那样要做 8 次. 由于 F^n 底行的第一个元素是 f_n, 下面过程能算出 f_n:

```
template<typename I>
    requires(Integer(I))
I fibonacci(I n)
{
    // 前条件: n ≥ 0
    if (n == I(0)) return I(0);
    return power(pair<I, I>(I(1), I(0)),
                 n,
                 fibonacci_matrix_multiply<I>).m0;
}
```

3.7 累积过程

前一章将动作定义为变换的一个对偶. 如果以下面语句的形式应用二元运算, 每个二元运算也存在一个对偶过程

```
x = op(x, y);
```

通过和另一对象结合的方式把一个对象送给二元运算, 用以修改其状态, 就定义了在该对象上的一个**累积过程** (accumulation procedure). 可以基于二元运算定义这种累积过程, 反之亦然:

```
void op_accumulate(T& x, const T& y) { x = op(x, y); }
  // 从二元运算定义累积过程
```

还有

```
T op(T x, const T& y) { op_accumulate(x, y); return x; }
  // 从累积过程定义二元运算
```

与动作的情况类似, 有时独立的实现更加高效, 在这种情况下需要同时提供二元运算和累积过程.

练习 3.3 请基于累积过程重写本章的算法.

项目 3.2 请基于 Miller and Brown [1966] 和 Fiduccia [1985] 的结果构造一个用于生成线性递归序列的库.

3.8 总结

如果一个算法可以应用于满足某些需求条件 (例如可结合性) 的不同模型, 它就是抽象的. 代码优化也基于等式推理. 除非已知有关的类型是规范的, 否则可以做的优化就非常少. 处理特殊情况的过程可能使代码更为高效甚至更抽象. 数学和抽象算法的结合可能导出令人惊异的算法, 例如在对数时间里生成线性递归序列里的第 n 个元素.

第 4 章

线性序

本章讨论二元关系的性质 (例如传递性和对称性), 并将着重介绍全序和弱线性序. 这里要引进基于线性序的函数的稳定性的概念, 稳定性保证维持作为参数的等价元素的序关系. 我们还要把 min 和 max 推广到序选择函数, 例如三个元素的中间值元素. 这里还要介绍一种技术, 它可以通过将问题归约到受限子问题的方式处理其实现的复杂性.

4.1 关系的分类

关系 (relation) 是有两个同类型参数的谓词:

$$\begin{aligned} &\mathit{Relation}(\mathsf{Op}) \triangleq \\ &\quad \mathit{HomogeneousPredicate}(\mathsf{Op}) \\ &\quad \wedge\ \mathsf{Arity}(\mathsf{Op}) = 2 \end{aligned}$$

一个关系是传递的 (transitive), 如果它对 a 和 b 成立且对 b 和 c 成立, 那么就一定对 a 和 c 成立:

$$\begin{aligned} &\textbf{property}(\mathsf{R} : \mathit{Relation}) \\ &\mathsf{transitive} : \mathsf{R} \\ &\quad \mathsf{r} \mapsto (\forall \mathsf{a}, \mathsf{b}, \mathsf{c} \in \mathsf{Domain}(\mathsf{R}))\,(\mathsf{r}(\mathsf{a}, \mathsf{b}) \wedge \mathsf{r}(\mathsf{b}, \mathsf{c}) \Rightarrow \mathsf{r}(\mathsf{a}, \mathsf{c})) \end{aligned}$$

传递关系的例子如相等、二元组的第一个元素相等、轨道上的可达性、整除关系等.

一个关系是严格的 (strict), 如果它绝不在任一个元素与其自身之间成立; 一个关系是自反的 (reflexive), 如果每个元素与其自身之间总有这一关系:

property(R : *Relation*)

strict : R

$r \mapsto (\forall a \in \mathsf{Domain}(R))\, \neg r(a, a)$

property(R : *Relation*)

reflexive : R

$r \mapsto (\forall a \in \mathsf{Domain}(R))\, r(a, a)$

前面作为例子的关系都是自反的, 真因子关系是严格的.

练习 4.1 请给出一个关系的例子, 它既不严格也不自反.

一个关系是**对称的**, 如果任何时候它在一个方向上成立, 也必然在另一方向上成立. 一个关系是**反对称的**, 如果它绝不会在两个方向上都成立:

property(R : *Relation*)

symmetric : R

$r \mapsto (\forall a, b \in \mathsf{Domain}(R))\, (r(a, b) \Rightarrow r(b, a))$

property(R : *Relation*)

asymmetric : R

$r \mapsto (\forall a, b \in \mathsf{Domain}(R))\, (r(a, b) \Rightarrow \neg r(b, a))$

对称关系的一个例子是 "兄弟姐妹"; 反对称关系的一个例子是 "长辈".

练习 4.2 请给出一个关系的例子, 它是对称的但不传递.

练习 4.3 请给出一个关系的例子, 它是对称的但不自反.

对给定的关系 $r(a, b)$, 存在一些具有相同作用域的**派生关系** (derived relation):

$$\begin{aligned} \mathsf{complement}_r(a, b) &\Leftrightarrow \neg r(a, b) \\ \mathsf{converse}_r(a, b) &\Leftrightarrow r(b, a) \\ \mathsf{complement\_of\_converse}_r(a, b) &\Leftrightarrow \neg r(b, a) \end{aligned}$$

对于对称关系而言, 有意思的派生关系只有它的补 (complement), 因为它的逆 (converse) 关系就等于它自身.

一个关系是一个**等价关系**, 如果它同时是传递的、自反的和对称的:

property(R : *Relation*)

equivalence : R

$r \mapsto$ transitive(r) $\wedge$ reflexive(r) $\wedge$ symmetric(r)

等价关系的例子包括相等、几何全等、整数的模 n 相等.

引理 4.1 如果 r 是等价关系, 那么 $a = b \Rightarrow r(a, b)$.

一个等价关系把它的定义域划分为一集*等价类* (equivalence class), 即划分为一些子集, 每个子集包含所有等价于某给定元素的所有元素. 我们通常可以通过定义一个*关键码函数* (key function) 的方式来实现一个等价关系. 这一关键码函数对每个等价类里的所有元素返回一个唯一值. 将相等判断应用于关键码函数的结果, 就能确定两个元素是否等价:

property(F : *UnaryFunction*, R : *Relation*)

requires(Domain(F) = Domain(R))

key_function : F × R

$(f, r) \mapsto (\forall a, b \in \text{Domain}(F))\,(r(a, b) \Leftrightarrow f(a) = f(b))$

引理 4.2 key_function(f, r) $\Rightarrow$ equivalence(r)

4.2 全序和弱序

一个关系是一个*全序* (total order), 如果它是传递的, 而且满足*三分律* (trichotomy law), 也就是说, 对任一对元素, 下面三者中恰有一个成立: 该关系, 该关系的逆, 或两者相等:

property(R : *Relation*)

total_ordering : R

$r \mapsto$ transitive(r) $\wedge$

$(\forall a, b \in \text{Domain}(R))$ 下面三者中恰有一个成立:

$r(a, b), r(b, a),$ 或 $a = b$

一个关系是一个*弱序* (weak ordering), 如果它是传递的, 而且在原定义域上存在一个等价关系, 使原关系满足*弱三分律* (weak trichotomy law), 即, 对任一对元素下面三者中恰有一个成立: 该关系, 该关系的逆, 或者上述等价关系:

property(R : *Relation*, E : *Relation*) **requires**(Domain(R) = Domain(E))

weak_ordering : R

$r \mapsto$ transitive(r) $\wedge$ $(\exists e \in E)$equivalence(e) $\wedge$

$(\forall a, b \in \text{Domain}(R))$ 下面三者中恰有一个成立:

$r(a,b), r(b,a)$, 或 $e(a,b)$

给定关系 r, 关系 $\neg r(a,b) \wedge \neg r(b,a)$ 称为 r 的对称补 (symmetric complement).

引理 4.3 弱序的对称补是一个等价关系.

弱序的例子如基于其第一个元排序的二元组, 或基于工资排序的雇员.

引理 4.4 全序是弱序.

引理 4.5 弱序是反对称的.

引理 4.6 弱序是严格的.

集合 T 上的一个关键元函数 f, 加上 f 的值域上的一个全序 r, 定义了 T 上的一个弱序 $\tilde{r}(x,y) \Leftrightarrow r(f(x), f(y))$.

下面把全序和弱序统称为线性序 (linear ordering), 因为它们都遵从三分律.

4.3 按序选取

给定弱序 r 和取自 r 的作用域的两个元素 a 和 b, 问两者中哪个更小就有意义. 当 r 或其逆对 a 和 b 成立时, 很容易定义两者中的较小元; 但如果两者等价, 事情就不好处理了. 考虑哪个元素较大时也会遇到类似问题.

处理这个问题的一个重要性质称为稳定性 (stability). 非形式地说, 一个算法是稳定的, 如果它不会破坏等价元素原来的顺序. 因此, 如果将取较小 (minimum) 或较大 (maximum) 看作从一个包含两个参数的表中选取较小或者较大元素, 在对两个等价的元素调用选取较小元操作时, 稳定性要求必须返回第一个参数; 而取较大元的操作则应返回第二个元素.[1]

可以把选取较小元和较大元推广到 (j, k)-序选取, 这里的 $k > 0$ 表示参数的个数, 而 $0 \leqslant j < k$ 表示要求选取第 j 个最小的元素. 为了形式化稳定性的概念,

1. 后面几章将把稳定性的概念扩展到其他算法类.

假定 k 个参数中的每个都关联着一个唯一自然数, 称为其稳定性索引 (stability index). 给定了原始的弱序 r 后, 我们将 (对象, 稳定性索引) 对偶上的增强关系 (strengthened relation) $\hat{r}$ 定义为:

$$\hat{r}((a, i_a), (b, i_b)) \Leftrightarrow r(a, b) \vee (\neg r(b, a) \wedge i_a < i_b)$$

如果基于 $\hat{r}$ 实现按序选取算法, 即使对等价元素, 这里也没有了歧义性. 一个参数的默认稳定性索引就是它在参数表里的位置.

增强关系 $\hat{r}$ 也是对稳定性进行推理的强大工具. 实际上, 很容易定义一个简单的按序选取过程, 而不显式表示稳定性索引. 下面的选较小元实现在 a 和 b 等价时返回 a, 它满足有关稳定性的定义:[2]

```
template<typename R>
    requires(Relation(R))
const Domain(R)& select_0_2(const Domain(R)& a,
                            const Domain(R)& b, R r)
{
    // 前条件: weak_ordering(r)
    if (r(b, a)) return b;
    return a;
}
```

类似的, 下面的选较大元实现在 a 和 b 等价时返回 b, 满足前面有关稳定性的定义:[3]

```
template<typename R>
    requires(Relation(R))
const Domain(R)& select_1_2(const Domain(R)& a,
                            const Domain(R)& b, R r)
{
    // 前条件: weak_ordering(r)
    if (r(b, a)) return a;
```

2. 有关命名方面的约定在本节后面解释.

3. STL 不正确地要求 max(a, b) 在 a 和 b 等价时返回 a.

```
    return b;
}
```

本章剩下部分的讨论中总假定有前条件 weak_ordering(r).

虽然对任意 k 个参数的按序选取过程都可能有用, 但是随着 k 值的增长, 写出这样的按序选取函数也会很快变得越来越困难, 而且会有太多的过程可能一次也不会被人使用. 有一种称为*归约到受限子问题* (reduction to constrained subproblem) 的技术可以很好地处理这两个问题. 下面要开发一族过程, 其中假定了一些与参数的相对序有关的信息.

这里为过程进行系统化命名的问题至关重要. 每个名字都以 select_j_k 开头, 其中 $0 \leqslant j < k$, 表明是从 k 个参数中选取第 j 个大的元素. 这里还要附加一系列字母来表示有关参数序的前条件. 例如, 后缀 _ab 意味着前两个参数有序, 而 _abd 意味着第 1, 2 和 4 个元素有序. 如果对不同的参数链都有前条件, 就用多个这样的后缀.

看几个例子. 取三个元素中的最小元和最大元很容易实现:

```
template<typename R>
    requires(Relation(R))
const Domain(R)& select_0_3(const Domain(R)& a,
                            const Domain(R)& b,
                            const Domain(R)& c, R r)
{
    return select_0_2(select_0_2(a, b, r), c, r);
}

template<typename R>
    requires(Relation(R))
const Domain(R)& select_2_3(const Domain(R)& a,
                            const Domain(R)& b,
                            const Domain(R)& c, R r)
{
    return select_1_2(select_1_2(a, b, r), c, r);
}
```

如果已知前两个元素是上升序, 很容易找出三个元素里的中间值元素:

```
template<typename R>
    requires(Relation(R))
const Domain(R)& select_1_3_ab(const Domain(R)& a,
                               const Domain(R)& b,
                               const Domain(R)& c, R r)
{
    if (!r(c, b)) return b;      // a, b, c 是按顺序排列的
    return select_1_2(a, c, r); // b 不是中间值元素
}
```

建立 select_1_3_ab 的前条件只需做一次比较. 因为参数通过常量引用传递, 而这里并不移动数据:

```
template<typename R>
    requires(Relation(R))
const Domain(R)& select_1_3(const Domain(R)& a,
                            const Domain(R)& b,
                            const Domain(R)& c, R r)
{
    if (r(b, a)) return select_1_3_ab(b, a, c, r);
    return              select_1_3_ab(a, b, c, r);
}
```

在最坏情况下 select_1_3 要做 3 次比较. 如果 c 是 a, b, c 中的最大元素, 函数就只需做两次比较. 由于发生后一情况有三分之一的可能性, 因此平均比较次数是 $2\frac{2}{3}$. 这里假定各种输入情况平均分布.

从 n 个元素里找出第二小的元素至少需要做 $n + \lceil \log_2 n \rceil - 2$ 次比较.[4] 特别的, 从 4 个元素中找出第二小的元素需要做 4 次比较.

如果知道参数中第一对元素和第二对元素都分别为上升排序的, 那么就很容易从中选出第二小的元素:

4. 这一结论是 Jozef Schreier 提出的猜想, 后来被 Sergei Kislitsyn 证明 [Knuth 1998, 209 页, Theorem S].

```
template<typename R>
    requires(Relation(R))
const Domain(R)& select_1_4_ab_cd(const Domain(R)& a,
                                  const Domain(R)& b,
                                  const Domain(R)& c,
                                  const Domain(R)& d, R r) {
    if (r(c, a)) return select_0_2(a, d, r);
    return               select_0_2(b, c, r);
}
```

如果已知第一对参数按上升序排列，再做一次比较就可以建立起select_1_4_ab_cd的前条件：

```
template<typename R>
    requires(Relation(R))
const Domain(R)& select_1_4_ab(const Domain(R)& a,
                               const Domain(R)& b,
                               const Domain(R)& c,
                               const Domain(R)& d, R r) {
    if (r(d, c)) return select_1_4_ab_cd(a, b, d, c, r);
    return               select_1_4_ab_cd(a, b, c, d, r);
}
```

做一次比较就可以建立select_1_4_ab的前条件：

```
template<typename R>
    requires(Relation(R))
const Domain(R)& select_1_4(const Domain(R)& a,
                            const Domain(R)& b,
                            const Domain(R)& c,
                            const Domain(R)& d, R r) {
    if (r(b, a)) return select_1_4_ab(b, a, c, d, r);
    return               select_1_4_ab(a, b, c, d, r);
}
```

练习 4.4 请实现 select_2_4.

维持直至 4 阶的按序选取网络的稳定性不太困难. 但到 5 阶就出现了新情况, 其中对应于一个受限子问题的过程被原调用方以不合顺序的参数调用了, 这就违背了稳定性. 处理这种情况的系统化方法是把稳定性索引和各实参一起传递, 并使用强化关系 $\hat{r}$. 下面利用整数模板参数来避免额外的运行时开销.

将对应于参数 a, b 等的稳定性索引命名为 ia, ib, ... 利用函数对象模板 compare_strict_or_reflexive 得到增强的关系 $\hat{r}$, 这个函数对象模板有一个类型为 bool 的模板参数. 该参数为真时, 意味着其参数的稳定性索引是按上升序排列的:

```
template<bool strict, typename R>
    requires(Relation(R))
struct compare_strict_or_reflexive;
```

在构造 compare_strict_or_reflexive 的实例时, 送给它了一个适当的布尔类型的模板实参:

```
template<int ia, int ib, typename R>
    requires(Relation(R))
const Domain(R)& select_0_2(const Domain(R)& a,
                            const Domain(R)& b, R r)
{
    compare_strict_or_reflexive<(ia < ib), R> cmp;
    if (cmp(b, a, r)) return b;
    return a;
}
```

下面要针对两种情况做出 compare_strict_or_reflexive 的: (1) 按上升序排列的稳定性索引, 在这种情况下用原来的严格关系 r; (2) 按下降序排列的稳定性索引, 这种情况下用关系 r 对应的自反版本:

```
template<typename R>
    requires(Relation(R))
struct compare_strict_or_reflexive<true, R> // 严格的
```

```
{
    bool operator()(const Domain(R)& a,
                    const Domain(R)& b, R r)
    {
        return r(a, b);
    }
};

template<typename R>
    requires(Relation(R))
struct compare_strict_or_reflexive<false, R> // 自反的
{
    bool operator()(const Domain(R)& a,
                    const Domain(R)& b, R r)
    {
        return !r(b, a); // complement_of_converse_r(a,b)
    }
};
```

当一个带有稳定性索引的序选择过程调用另一这类过程时,稳定性索引要与参数对应,按参数在调用中出现的同样顺序传递它们:

```
template<int ia, int ib, int ic, int id, typename R>
    requires(Relation(R))
const Domain(R)& select_1_4_ab_cd(const Domain(R)& a,
                                  const Domain(R)& b,
                                  const Domain(R)& c,
                                  const Domain(R)& d, R r)
{
    compare_strict_or_reflexive<(ia < ic), R> cmp;
    if (cmp(c, a, r)) return
        select_0_2<ia,id>(a, d, r);
    return
```

```
        select_0_2<ib,ic>(b, c, r);
}

template<int ia, int ib, int ic, int id, typename R>
    requires(Relation(R))
const Domain(R)& select_1_4_ab(const Domain(R)& a,
                               const Domain(R)& b,
                               const Domain(R)& c,
                               const Domain(R)& d, R r)
{
    compare_strict_or_reflexive<(ic < id), R> cmp;
    if (cmp(d, c, r)) return
        select_1_4_ab_cd<ia,ib,id,ic>(a, b, d, c, r);
    return
        select_1_4_ab_cd<ia,ib,ic,id>(a, b, c, d, r);
}

template<int ia, int ib, int ic, int id, typename R>
    requires(Relation(R))
const Domain(R)& select_1_4(const Domain(R)& a,
                            const Domain(R)& b,
                            const Domain(R)& c,
                            const Domain(R)& d, R r)
{
    compare_strict_or_reflexive<(ia < ib), R> cmp;
    if (cmp(b, a, r)) return
        select_1_4_ab<ib,ia,ic,id>(b, a, c, d, r);
    return
        select_1_4_ab<ia,ib,ic,id>(a, b, c, d, r);
}
```

至此我们已经为实现 5 选择做好了准备:

```
template<int ia, int ib, int ic, int id, int ie, typename R>
    requires(Relation(R))
const Domain(R)& select_2_5_ab_cd(const Domain(R)& a,
                                  const Domain(R)& b,
                                  const Domain(R)& c,
                                  const Domain(R)& d,
                                  const Domain(R)& e, R r)
{
    compare_strict_or_reflexive<(ia < ic), R> cmp;
    if (cmp(c, a, r)) return
        select_1_4_ab<ia,ib,id,ie>(a, b, d, e, r);
    return
        select_1_4_ab<ic,id,ib,ie>(c, d, b, e, r);
}

template<int ia, int ib, int ic, int id, int ie, typename R>
    requires(Relation(R))
const Domain(R)& select_2_5_ab(const Domain(R)& a,
                               const Domain(R)& b,
                               const Domain(R)& c,
                               const Domain(R)& d,
                               const Domain(R)& e, R r)
{
    compare_strict_or_reflexive<(ic < id), R> cmp;
    if (cmp(d, c, r)) return
        select_2_5_ab_cd<ia,ib,id,ic,ie>(a, b, d, c, e, r);
    return
        select_2_5_ab_cd<ia,ib,ic,id,ie>(a, b, c, d, e, r);
}

template<int ia, int ib, int ic, int id, int ie, typename R>
    requires(Relation(R))
```

```
const Domain(R)& select_2_5(const Domain(R)& a,
                            const Domain(R)& b,
                            const Domain(R)& c,
                            const Domain(R)& d,
                            const Domain(R)& e, R r)
{
    compare_strict_or_reflexive<(ia < ib), R> cmp;
    if (cmp(b, a, r)) return
        select_2_5_ab<ib,ia,ic,id,ie>(b, a, c, d, e, r);
    return
        select_2_5_ab<ia,ib,ic,id,ie>(a, b, c, d, e, r);
}
```

引理 4.7 select_2_5 执行 6 次比较.

练习 4.5 请设法找出一个算法, 它返回 5 个参数的中间值元素, 但做的比较次数比平均情况少一点.

我们可以把按序选取过程包装在一个外围过程里, 该外围过程可以提供任何严格递增的整数常数序列作为稳定性索引. 为方便起见, 这里直接用从 0 开始的顺序整数:

```
template<typename R>
    requires(Relation(R))
const Domain(R)& median_5(const Domain(R)& a,
                          const Domain(R)& b,
                          const Domain(R)& c,
                          const Domain(R)& d,
                          const Domain(R)& e, R r)
{
    return select_2_5<0,1,2,3,4>(a, b, c, d, e, r);
}
```

练习 4.6 请证明本节里的每个按序选取过程的稳定性.

练习 4.7 通过穷尽测试验证本节里每个按序选取过程的正确性和稳定性.

项目 4.1 请为保证按序选取过程的组合的稳定性提出一集充分必要条件.

项目 4.2 请为稳定的排序和归并创建一个最大最小过程库.[5] 不仅使比较的次数达到最小, 也使数据移动次数达到最小.

4.4 自然全序

在一个类型里, 两个值相等意味着它们表示同一实体, 所以, 在任何类型上只有唯一的一个相等关系. 但在一个类型上未必只有唯一的一个自然全序. 对一个具体类别, 经常存在多个全序和弱序, 它们中并没有哪个扮演特殊的角色. 而对一个抽象类别, 却可能有一个符合其基本运算需要的特殊全序. 这个序称为它的**自然全序** (natural total ordering), 用符号 $<$ 表示, 描述如下:

$$
\begin{aligned}
&\mathit{TotallyOrdered}(\mathsf{T}) \triangleq \\
&\quad \mathit{Regular}(\mathsf{T}) \\
&\quad \wedge\ <: \mathsf{T} \times \mathsf{T} \to \mathsf{bool} \\
&\quad \wedge\ \mathsf{total\_ordering}(<)
\end{aligned}
$$

例如, 整数的自然全序符合其基本运算的需要:

$$
\begin{aligned}
&a < \mathsf{successor}(a) \\
&a < b \Rightarrow \mathsf{successor}(a) < \mathsf{successor}(b) \\
&a < b \Rightarrow a + c < b + c \\
&a < b \wedge 0 < c \Rightarrow ca < cb
\end{aligned}
$$

有些类型并没有自然全序. 例如复数类型和雇员记录都没有. 我们要求任意一个规范类型都提供一个**默认全序** (default total ordering. 有时简称为**默认序**, default ordering), 以便能做对数时间的检索. 作为将非自然全序看作默认序的例子, 可以考虑复数上的字典序. 如果存在自然全序, 它就是默认序. 我们采用下面的记法:

	规程	C++
T 的默认序	$\mathsf{less}_{\mathsf{T}}$	`less<T>`

5. 参看 Knuth [1998, 5.3 节: 最优排序].

4.5 派生过程组

有些过程很自然地成组出现, 只要定义了该组中的某个过程, 自然就有了整个组里的其他过程. 相等的补就是不等, 只要有相等它就自然有了定义, 因此运算符 $=$ 和 $\neq$ 必须协调地定义. 对每个全序类型, 4 个运算符 $<$, $>$, $\leqslant$ 和 $\geqslant$ 必须协调地定义, 以保证下面几个关系成立:

$$a > b \Leftrightarrow b < a$$
$$a \leqslant b \Leftrightarrow \neg(b < a)$$
$$a \geqslant b \Leftrightarrow \neg(a < b)$$

4.6 按序选取过程的扩展

本章讨论的按序选取过程返回的对象是不能变动的, 因为过程里用的都是常量引用. 可以定义另一种过程版本返回可变动的对象, 使这种对象可以用在赋值的左边, 或者作为动作或累积过程的可变动参数. 这样的过程版本很有用, 定义也直截了当. 可以用重载的方式实现按序选取过程的变动版本, 为此只需从非变动版本中简单地去掉每个参数类型和返回值类型的 const. 看一个例子, 前面的 select_0_2 的一个新版本是

```
template<typename R>
    requires(Relation(R))
Domain(R)& select_0_2(Domain(R)& a, Domain(R)& b, R r)
{
    if (r(b, a)) return b;
    return a;
}
```

进一步说, 实现全序类型 (有 $<$) 的库应该提供这些版本的过程, 因为它们都经常需要. 这意味着每个过程需要 4 个不同版本.

全序或弱序满足三分律或弱三分律, 这提示我们可以不用二值关系, 而用一种三值的比较过程 . 在一些情况下这样做可能避免一次额外的过程调用.

练习 4.8 请用三值比较过程重写本章的各算法.

4.7 总结

全序和弱序的公理为联系特定的序和通用算法提供了一个接口. 对一些小问题的系统化的解, 可以有利于简化大问题的分解. 存在许多具有相互关联的语义的过程组.

第 5 章

有序代数结构

本章讨论来自抽象代数的一些概念的层次结构, 从半群开始直至环和模. 而后用全序的观点组合这些代数概念. 当有序的代数结构具有阿基米德性质时, 就可以为之定义高效的找出商和余的算法, 而这种商和余又能用于导出求最大公因子的广义欧几里得算法. 对一些相关的逻辑概念, 如协调性和独立性, 这里只做简单处理, 最后用有关计算机整数算术的讨论作为总结.

5.1 基本代数结构

一个元素称为某二元运算的*单位元* (identity element), 如果它与任意的另一个元素组合时, 得到的结果总是那个元素:

property(T : *Regular*, Op : *BinaryOperation*)
  requires(T = Domain(Op))
identity_element : T × Op
  $(e, \mathrm{op}) \mapsto (\forall a \in \mathsf{T})\ \mathrm{op}(a, e) = \mathrm{op}(e, a) = a$

引理 5.1 单位元唯一:

$$\mathsf{identity\_element}(e, \mathrm{op}) \wedge \mathsf{identity\_element}(e', \mathrm{op}) \Rightarrow e = e'$$

空串是串拼接运算的单位元. 矩阵 $\left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right)$ 是 2×2 矩阵的乘法的单位元, 而 $\left(\begin{smallmatrix}0&0\\0&0\end{smallmatrix}\right)$ 是 2×2 矩阵的加法的单位元.

一个变换称为某二元运算相对于一个给定元素 (通常取该二元运算的单位元) 的*逆运算* (inverse operation), 如果该变换满足下面的性质:

property(F : *Transformation*, T : *Regular*, Op : *BinaryOperation*)

requires(Domain(F) = T = Domain(Op))

inverse_operation : F × T × Op

$(inv, e, op) \mapsto (\forall a \in T)\, op(a, inv(a)) = op(inv(a), a) = e$

引理 5.2 n^3 是正整数 $n \neq 0$ 模 5 的乘法逆.

一个二元运算**可交换** (commutative), 如果其参数交换位置后结果不变:

property(Op : *BinaryOperation*)

commutative : Op

$op \mapsto (\forall a, b \in \mathsf{Domain}(Op))\, op(a, b) = op(b, a)$

显然, 函数复合是可结合的但不是可交换的.

带有一个可交换运算的集合称为一个**半群** (semigroup). 如第 3 章所说, 我们总用 + 表示一个可结合且可交换的运算, 因此把带有 + 运算的类型称为一个**加半群** (additive semigroup):

AdditiveSemigroup(T) ≙

Regular(T)

∧ + : T × T → T

∧ associative(+)

∧ commutative(+)

有的乘法不是可交换的, 矩阵乘法就是一个例子.

MultiplicativeSemigroup(T) ≙

Regular(T)

∧ · : T × T → T

∧ associative(·)

我们使用如下记法:

	规程	C++
乘法	·	*

带有单位元的半群称为**幺半群** (monoid). 加法的单位元用 0 表示, 由此得到**加法幺半群** (additive monoid) 的定义:

AdditiveMonoid(T) ≜

AdditiveSemigroup(T)

∧ 0 ∈ T

∧ identity_element(0, +)

我们使用如下记法:

	规程	C++
加法单位元	0	T(0)

非负实数是一个加法幺半群, 以自然数为系数的矩阵也是.

乘法的单位元用 1 表示, 由此可以得到**乘法幺半群** (multiplicative monoid) 的定义:

MultiplicativeMonoid(T) ≜

MultiplicativeSemigroup(T)

∧ 1 ∈ T

∧ identity_element(1, ·)

我们使用如下记法:

	规程	C++
乘法单位元	1	T(1)

整数系数矩阵是一个乘法幺半群.

带有逆运算的幺半群称为**群** (group). 如果一个加法幺半群有逆运算, 就用一元的 − 表示它, 由它可以得到一个称为减的派生运算, 用二元 − 表示. 这样就得到了**加法群** (additive group) 的定义:

AdditiveGroup(T) ≜

AdditiveMonoid(T)

∧ − : T → T

∧ inverse_operation(unary −, 0, +)

∧ − : T × T → T

(a, b) ↦ a + (−b)

整数系数矩阵是一个加法群.

引理 5.3 在加法群里 $-0 = 0$.

正如可以有加法群的概念一样, 存在与之对应的乘法群 (multiplicative group) 的概念. 其中的逆称为乘法逆, 由它得到的二元派生运算称为除法 (division), 用二元 / 表示:

$$
\begin{aligned}
&\mathit{MultiplicativeGroup}(\mathsf{T}) \triangleq \\
&\quad \mathit{MultiplicativeMonoid}(\mathsf{T}) \\
&\wedge\ \mathsf{multiplicative\_inverse} : \mathsf{T} \to \mathsf{T} \\
&\wedge\ \mathsf{inverse\_operation}(\mathsf{multiplicative\_inverse}, 1, \cdot) \\
&\wedge\ / : \mathsf{T} \times \mathsf{T} \to \mathsf{T} \\
&\qquad (a, b) \mapsto a \cdot \mathsf{multiplicative\_inverse}(b)
\end{aligned}
$$

multiplicative_inverse(x) 记为 x^{-1}.

单位圆上的复数 $\{\cos\theta + i\sin\theta\}$ 是一个可交换的乘法群. 幺模群 $\mathrm{GL}_n(\mathbb{Z})$ (行列式等于 ± 1 的 $n \times n$ 整系数矩阵的群) 是一个不可交换的乘法群.

同一类型上的两个概念可以用关于它们的运算的公理组合起来. 当一个类型上同时有 $+$ 和 $\cdot$ 运算时, 通过公理将其连接可以定义半环 (semiring):

$$
\begin{aligned}
&\mathit{Semiring}(\mathsf{T}) \triangleq \\
&\quad \mathit{AdditiveMonoid}(\mathsf{T}) \\
&\wedge\ \mathit{MultiplicativeMonoid}(\mathsf{T}) \\
&\wedge\ 0 \neq 1 \\
&\wedge\ (\forall a \in \mathsf{T})\, 0 \cdot a = a \cdot 0 = 0 \\
&\wedge\ (\forall a, b, c \in \mathsf{T}) \\
&\qquad a \cdot (b + c) = a \cdot b + a \cdot c \\
&\qquad \wedge\ (b + c) \cdot a = b \cdot a + c \cdot a
\end{aligned}
$$

有关乘以 0 的公理称为零化性质 (annihilation property). 关联 $+$ 和 $\cdot$ 的最后一个性质称为分配律 (distributive).

具有非负整系数的矩阵构成一个半环.

$$
\begin{aligned}
&\mathit{CommutativeSemiring}(\mathsf{T}) \triangleq \\
&\quad \mathit{Semiring}(\mathsf{T}) \\
&\wedge\ \mathsf{commutative}(\cdot)
\end{aligned}
$$

非负整数构成一个交换半环.

$$
\begin{aligned}
&\mathit{Ring}(\mathsf{T}) \triangleq \\
&\qquad \mathit{AdditiveGroup}(\mathsf{T}) \\
&\quad \wedge\ \mathit{Semiring}(\mathsf{T})
\end{aligned}
$$

整系数矩阵构成一个环 (ring).

$$
\begin{aligned}
&\mathit{CommutativeRing}(\mathsf{T}) \triangleq \\
&\qquad \mathit{AdditiveGroup}(\mathsf{T}) \\
&\quad \wedge\ \mathit{CommutativeSemiring}(\mathsf{T})
\end{aligned}
$$

整数构成一个交换环 (commutative ring); 整系数多项式构成一个交换环.

关系概念 (relational concept) 是定义在两个类型上的概念. 半模 (semimodule) 是一个关系概念, 它连接一个加法幺半群和一个交换半环:

$$
\begin{aligned}
&\mathit{Semimodule}(\mathsf{T},\mathsf{S}) \triangleq \\
&\qquad \mathit{AdditiveMonoid}(\mathsf{T}) \\
&\quad \wedge\ \mathit{CommutativeSemiring}(\mathsf{S}) \\
&\quad \wedge\ \cdot : \mathsf{S} \times \mathsf{T} \to \mathsf{T} \\
&\quad \wedge\ (\forall \alpha, \beta \in \mathsf{S})(\forall a, b \in \mathsf{T}) \\
&\qquad\qquad \alpha \cdot (\beta \cdot a) = (\alpha \cdot \beta) \cdot a \\
&\qquad\qquad (\alpha + \beta) \cdot a = \alpha \cdot a + \beta \cdot a \\
&\qquad\qquad \alpha \cdot (a + b) = \alpha \cdot a + \alpha \cdot b \\
&\qquad\qquad 1 \cdot a = a
\end{aligned}
$$

如果有 $\mathit{Semimodule}(\mathsf{T},\mathsf{S})$, 就说 T 是 S 上的一个半模. 这里借用向量空间的术语, 称 T 的元素为向量 (vector), S 的元素为标量 (scalar). 举例说, 非负整数系数的多项式构成了非负整数上的一个半模.

定理 5.1 $\mathit{AdditiveMonoid}(\mathsf{T}) \Rightarrow \mathit{Semimodule}(\mathsf{T},\mathbb{N})$, 这里的标量乘法定义为 $n \cdot x = \underbrace{x + \cdots + x}_{n\text{ 个}}$.

证明. 由标量乘法的定义, 幺半群运算的可结合性和可交换性直接可得. 例如,

$$\begin{aligned} n \cdot a + n \cdot b &= (a + \cdots + a) + (b + \cdots + b) \\ &= (a + b) + \cdots + (a + b) \\ &= n \cdot (a + b) \end{aligned}$$

□

利用第 3 章的 power, 可以实现在 $\log_2 n$ 步数内完成乘一个整数的算法.

通过用加法群代替加法幺半群, 用环代替半环的方式强化需求, 可以把半模变换为模:

$$\begin{aligned} &Module(T, S) \triangleq \\ &\quad Semimodule(T, S) \\ &\wedge\ AdditiveGroup(T) \\ &\wedge\ Ring(S) \end{aligned}$$

引理 5.4 每个加法群是整数上的一个模, 它带有一个适当定义的标量乘法.

计算机里的一个类型通常是某个概念的一个部分模型. 一个模型称为是**部分的** (partial), 如果相关的运算在有定义的地方满足公理, 但它们并不是处处有定义. 例如, 由于存储有限, 串拼接的结果可能无法表示, 但只要结果有定义, 拼接运算都可结合.

5.2 有序代数结构

如果在一个结构上定义了一个全序, 而且此全序与该结构的代数性质协调, 就称其为该结构上的一个**自然全序** (natural total ordering):

$$\begin{aligned} &OrderedAdditiveSemigroup(T) \triangleq \\ &\quad AdditiveSemigroup(T) \\ &\wedge\ TotallyOrdered(T) \\ &\wedge\ (\forall a, b, c \in T)\ a < b \Rightarrow a + c < b + c \end{aligned}$$

$$\begin{aligned} &OrderedAdditiveMonoid(T) \triangleq \\ &\quad OrderedAdditiveSemigroup(T) \\ &\wedge\ AdditiveMonoid(T) \end{aligned}$$

$$\begin{aligned}&\mathit{OrderedAdditiveGroup}(\mathsf{T}) \triangleq \\ &\qquad \mathit{OrderedAdditiveMonoid}(\mathsf{T}) \\ &\quad \wedge\ \mathit{AdditiveGroup}(\mathsf{T})\end{aligned}$$

引理 5.5 对有序的加法半群, 总有 $a < b \wedge c < d \Rightarrow a + c < b + d$.

引理 5.6 对一个可以看作自然数上的半模的有序加法幺半群, 总有 $a > 0 \wedge n > 0 \Rightarrow na > 0$.

引理 5.7 对有序加法群, 总有 $a < b \Rightarrow -b < -a$.

全序和取负运算使我们可以定义**绝对值** (absolute value):

```
template<typename T>
    requires(OrderedAdditiveGroup(T))
T abs(const T& a)
{
    if (a < T(0)) return -a;
    else          return  a;
}
```

下面引理说明了 abs 的一个重要性质.

引理 5.8 在一个有序加法群里 $a < 0 \Rightarrow 0 < -a$.

我们用 $|a|$ 表示 a 的绝对值. 绝对值满足下面性质:

引理 5.9

$$\begin{aligned}|a - b| &= |b - a| \\ |a + b| &\leqslant |a| + |b| \\ |a - b| &\geqslant |a| - |b| \\ |a| = 0 &\Rightarrow a = 0 \\ a \neq 0 &\Rightarrow |a| > 0\end{aligned}$$

5.3 求余

可以看到, 在一个加法幺半群里反复做加法就得到非负整数倍的乘法. 在加法群里这一算法可以逆行, 即通过元素的反复减得到 $a = nb$ 形式的商, 表示用 b 除 a. 把这种商扩展到任意元素之间的带余除法, 需要有一个序的概念. 该序使算法可以在不能再减时终止. 下面将看到, 它还使算法可以只需对数步就能完成. 减运算并不需要处处有定义, 只需要一个称为消除 (cancelation) 的部分减法就可以了, 其中的 $a - b$ 只在 b 不超过 a 时有定义:

$$
\begin{aligned}
&\mathit{CancellableMonoid}(\mathsf{T}) \triangleq \\
&\qquad \mathit{OrderedAdditiveMonoid}(\mathsf{T}) \\
&\quad \wedge\ - : \mathsf{T} \times \mathsf{T} \to \mathsf{T} \\
&\quad \wedge\ (\forall a, b \in \mathsf{T})\, b \leqslant a \Rightarrow a - b \text{ 有定义} \wedge (a - b) + b = a
\end{aligned}
$$

这里把公理写成 $(a - b) + b = a$ 而不是 $(a + b) - b = a$, 是为了避免部分模型 *CancellableMonoid* 出现溢出:

```
template<typename T>
    requires(CancellableMonoid(T))
T slow_remainder(T a, T b)
{
    // 前条件: a ≥ 0 ∧ b > 0
    while (b <= a) a = a - b;
    return a;
}
```

概念 *CancellableMonoid* 不够强, 不足以证明 slow_remainder 终止. 例如, 对于项按字典序排列的整系数多项式, slow_remainder 并不总终止.

练习 5.1 请给出两个整系数多项式实例, 上述算法对它们不终止.

要保证算法终止, 还需要另一个称为阿基米德公理 (Axiom of Archimedes) 的性质:[1]

1. "...(两个) 不相等的面积中较大者超出较小者的超出部分反复自加, 可以超过任何有限的面积." 见 Heath [1912, 234 页].

$$\begin{aligned}&\mathit{ArchimedeanMonoid}(\mathsf{T}) \triangleq\\ &\quad \mathit{CancellableMonoid}(\mathsf{T})\\ &\wedge\ (\forall \mathtt{a}, \mathtt{b} \in \mathsf{T})\,(\mathtt{a} \geqslant 0 \wedge \mathtt{b} > 0) \Rightarrow \mathsf{slow\_remainder}(\mathtt{a}, \mathtt{b}) \text{ 终止}\\ &\wedge\ \mathsf{QuotientType} : \mathit{ArchimedeanMonoid} \to \mathit{Integer}\end{aligned}$$

可以看到这一算法的终止是一个正当的公理, 在目前情况下它等价于

$$(\exists \mathtt{n} \in \mathsf{QuotientType}(\mathsf{T}))\ \mathtt{a} - \mathtt{n} \cdot \mathtt{b} < \mathtt{b}$$

阿基米德公理通常被说成 "存在整数 n 使得 $\mathtt{a} < \mathtt{n} \cdot \mathtt{b}$," 上述定义对部分的阿基米德幺半群 (Archimedean monoid) 有效, 其中 $\mathtt{n} \cdot \mathtt{b}$ 可能溢出. 类型函数 QuotientType 返回一个足以表示 slow_remainder 所执行的迭代次数的类型.

引理 5.10 下面是一些阿基米德幺半群的例子: 整数、有理数、二幂分数 $\{\frac{\mathtt{n}}{2^{\mathtt{k}}}\}$、三幂分数 $\{\frac{\mathtt{n}}{3^{\mathtt{k}}}\}$ 和实数.

很容易修改 slow_remainder 的代码, 要求它返回商:

```
template<typename T>
    requires(ArchimedeanMonoid(T))
QuotientType(T) slow_quotient(T a, T b)
{
    // 前条件: a ⩾ 0 ∧ b > 0
    QuotientType(T) n(0);
    while (b <= a) {
        a = a - b;
        n = successor(n);
    }
    return n;
}
```

反复加倍可以得到对数复杂性的 power 算法. 与之相关的另一个算法可以求出余数.[2] 基于 a 除以 2b 的余数 v, 可以推导出下面这个描述 a 除以 2b 的余数 u 的表达式:

$$\mathtt{a} = \mathtt{n}(2\mathtt{b}) + \mathtt{v}$$

2. 古埃及人用这一算法做带余除法, 也用求幂算法做乘法. 见 Robins and Shute [1987, 18 页].

由于 v 必须小于除数 2b, 所以

$$u = \begin{cases} v & \text{if } v < b \\ v - b & \text{if } v \geqslant b \end{cases}$$

这就给出了下面的递归过程:

```
template<typename T>
    requires(ArchimedeanMonoid(T))
T remainder_recursive(T a, T b)
{
    // 前条件: a ⩾ b > 0
    if (a - b >= b) {
        a = remainder_recursive(a, b + b);
        if (a < b) return a;
    }
    return a - b;
}
```

检查 $a - b \geqslant b$ 而不是 $a \geqslant b + b$ 可以避免 $b + b$ 溢出.

```
template<typename T>
    requires(ArchimedeanMonoid(T))
T remainder_nonnegative(T a, T b)
{
    // 前条件: a ⩾ 0 ∧ b > 0
    if (a < b) return a;
    return remainder_recursive(a, b);
}
```

练习 5.2 请分析 remainder_nonnegative 的复杂性.

[Floyd and Knuth 1990]提出了一个常量空间的阿基米德幺半群的求余算法, 它比 remainder_nonnegative 大约多执行 31% 的运算. 而如果可以用除 2 运算,

就存在不增加运算的算法.[3] 似乎在许多情景中都有这种现象. 例如, 虽然用直尺和圆规不可能做一个角的一般性 k-分式, 但做两分却极其简单.

$HalvableMonoid(\mathsf{T}) \triangleq$

$\quad ArchimedeanMonoid(\mathsf{T})$

$\quad \wedge\ \mathsf{half} : \mathsf{T} \to \mathsf{T}$

$\quad \wedge\ (\forall a, b \in \mathsf{T})\,(b > 0 \wedge a = b + b) \Rightarrow \mathsf{half}(a) = b$

请注意, half 只需对 "偶数" 元素有定义.

```
template<typename T>
    requires(HalvableMonoid(T))
T remainder_nonnegative_iterative(T a, T b)
{
    // 前条件: a ⩾ 0 ∧ b > 0
    if (a < b) return a;
    T c = largest_doubling(a, b);
    a = a - c;
    while (c != b) {
        c = half(c);
        if (c <= a) a = a - c;
    }
    return a;
}
```

其中的过程 largest_doubling 定义如下:

```
template<typename T>
    requires(ArchimedeanMonoid(T))
T largest_doubling(T a, T b)
{
    // 前条件: a ⩾ b > 0
    while (b <= a - b) b = b + b;
```

3. Dijkstra [1972, 13 页] 将此算法归功于 N. G. de Bruijn.

```
    return b;
}
```

remainder_nonnegative_iterative 的正确性依赖于下面引理.

引理 5.11 对可二分幺半群的正元素加倍 k 次, 得到的结果也可以折半 k 次.

对非可二分的阿基米德幺半群就只需要 remainder_nonnegative. 前面给出的实例 (欧几里得几何里的线段、有理数、二幂分数和三幂分数) 都是可二分的.

项目 5.1 是否存在是阿基米德幺半群, 但不是可二分幺半群的有用模型?

5.4 最大公因子

对阿基米德幺半群 T 里的 $a \geqslant 0$ 和 $b > 0$, **可整除性** (divisibility) 定义如下:

$$b \text{ 整除 } a \Leftrightarrow (\exists n \in \mathsf{QuotientType}(T))\, a = nb$$

引理 5.12 对于阿基米德幺半群 T 里的正元素 x, a, b:

- b 整除 $a \Leftrightarrow \mathsf{remainder\_nonnegative}(a, b) = 0$
- b 整除 $a \Rightarrow b \leqslant a$
- $a > b \wedge x$ 整除 $a \wedge x$ 整除 $b \Rightarrow x$ 整除 $(a - b)$
- x 整除 $a \wedge x$ 整除 $b \Rightarrow x$ 整除 $\mathsf{remainder\_nonnegative}(a, b)$

a 和 b 的**最大公因子** (greatest common divisor) 用 $\gcd(a, b)$ 表示. 它是 a 和 b 的因子, 而且能被 a 和 b 任何公因子整除.[4]

引理 5.13 在阿基米德幺半群里, 下面性质对正元素 x, a, b 成立:

- gcd 可交换
- gcd 可结合

4. 虽然这一定义对阿基米德幺半群有效, 但它并不依赖于序关系, 因此可以扩展到其他带有可整除关系的结构, 例如环.

- x 整除 a ∧ x 整除 b ⇒ x ⩽ gcd(a, b)
- gcd(a, b) 唯一
- gcd(a, a) = a
- a > b ⇒ gcd(a, b) = gcd(a − b, b)

由前面引理立即可知, 如果下面算法终止, 它将返回参数的 gcd:[5]

```
template<typename T>
    requires(ArchimedeanMonoid(T))
T subtractive_gcd_nonzero(T a, T b)
{
    // 前条件: a > 0 ∧ b > 0
    while (true) {
        if (b < a)      a = a - b;
        else if (a < b) b = b - a;
        else            return a;
    }
}
```

引理 5.14 对整数和有理数, 上面算法总能终止.

也存在一些类型, 上述算法对于它们并不总终止. 例如对于实数它就不一定终止, 比如对输入 $\sqrt{2}$ 和 1 不会终止. 这一事实的证明依赖于下面两个引理:

引理 5.15 $\gcd(\frac{a}{\gcd(a,b)}, \frac{b}{\gcd(a,b)}) = 1$

引理 5.16 如果整数 n 的平方根是偶数, 那么 n 也是偶数.

定理 5.2 subtractive_gcd_nonzero$(\sqrt{2}, 1)$ 不终止.

证明. 假设 subtractive_gcd_nonzero$(\sqrt{2}, 1)$ 终止并返回 d. 令 $m = \frac{\sqrt{2}}{d}$ 和 $n = \frac{1}{d}$. 由引理 5.15, m 和 n 没有大于 1 的公因子. 由 $\frac{m}{n} = \frac{\sqrt{2}}{1} = \sqrt{2}$, 所以 $m^2 = 2n^2$; m 是

5. 这个算法称为欧几里得算法 [Heath 1925, 第 3 卷, 14–22 页].

偶数; 对某个整数 u, $m = 2u$. 由于 $4u^2 = 2n^2$, 所以 $n^2 = 2u^2$; n 是偶数. 这样 m 和 n 都能被 2 整除; 矛盾.[6] □

一个*欧几里得幺半群* (Euclidean monoid) 是一个阿基米德幺半群, 如果在其中做 subtractive_gcd_nonzero 必定终止:

$$\begin{aligned}&\mathit{EuclideanMonoid}(\mathsf{T}) \triangleq\\ &\quad \mathit{ArchimedeanMonoid}(\mathsf{T})\\ &\quad \wedge\ (\forall a, b \in \mathsf{T})\,(a > 0 \wedge b > 0) \Rightarrow \mathsf{subtractive\_gcd\_nonzero}(a, b)\ \text{终止}\end{aligned}$$

引理 5.17 每个有最小正元素的阿基米德幺半群都是欧几里得的.

引理 5.18 有理数是一个欧几里得幺半群.

很容易扩充 subtractive_gcd_nonzero, 使之可以处理一个参数是 0 的情况, 因为任何 $b \neq 0$ 都整除该幺半群的 0 元:

```
template<typename T>
    requires(EuclideanMonoid(T))
T subtractive_gcd(T a, T b)
{
    // 前条件: a ⩾ 0 ∧ b ⩾ 0 ∧ ¬(a = 0 ∧ b = 0)
    while (true) {
        if (b == T(0)) return a;
        while (b <= a) a = a - b;
        if (a == T(0)) return b;
        while (a <= b) b = b - a;
    }
}
```

subtractive_gcd 里每个内层的 while 语句等价于对 slow_remainder 的一次调用. 利用前面的对数时间求余算法, 在 a 和 b 量级差异很大时的操作可以大大加速. 算法只依靠类型 T 上最基本的减法:

6. 一个正方形的边和对角线不可公度, 这是希腊人发现的最早证明之一. 亚里士多德 在 *Prior Analytics* I. 23 里将它作为通过反证法 (reductio ad absurdum) 证明的范例.

```
template<typename T>
    requires(EuclideanMonoid(T))
T fast_subtractive_gcd(T a, T b)
{
    // 前条件: a ≥ 0 ∧ b ≥ 0 ∧ ¬(a = 0 ∧ b = 0)
    while (true) {
        if (b == T(0)) return a;
        a = remainder_nonnegative(a, b);
        if (a == T(0)) return b;
        b = remainder_nonnegative(b, a);
    }
}
```

欧几里得幺半群的概念是对欧几里得算法的一种抽象, 它只依靠反复做减法.

5.5 广义 gcd

对整数可以用 fast_subtractive_gcd, 因为整数构成一个欧几里得幺半群. 对整数还可以用系统内部的求余数运算而不用 remainder_nonnegative. 进一步说, 这一算法对某些非阿基米德域也可以工作, 只要相应的域有一个合适的求余函数, 标准的长除算法很容易从十进制整数扩展到实数上的多项式.[7] 利用这样的求余函数就能计算两个多项式的 gcd 了.

与欧几里得算法相对应, 抽象代数还引进了欧几里得半环 (Euclidean semiring) 的概念 (也称为欧几里得域).[8] 实际上, 要求半环就足够了:

$$
\begin{aligned}
&\mathit{EuclideanSemiring}(\mathsf{T}) \triangleq \\
&\qquad \mathit{CommutativeSemiring}(\mathsf{T}) \\
&\quad \wedge\ \mathsf{NormType} : \mathit{EuclideanSemiring} \to \mathit{Integer} \\
&\quad \wedge\ \mathsf{w} : \mathsf{T} \to \mathsf{NormType}(\mathsf{T})
\end{aligned}
$$

7. 见 Chrystal [1904, 第 5 章].

8. 见 van der Waerden [1930, 第 3 章, 第 18 节].

$$\wedge\ (\forall a \in T)\, w(a) \geqslant 0$$
$$\wedge\ (\forall a \in T)\, w(a) = 0 \Leftrightarrow a = 0$$
$$\wedge\ (\forall a, b \in T)\, b \neq 0 \Rightarrow w(a \cdot b) \geqslant w(a)$$
$$\wedge\ \mathsf{remainder} : T \times T \to T$$
$$\wedge\ \mathsf{quotient} : T \times T \to T$$
$$\wedge\ (\forall a, b \in T)\, b \neq 0 \Rightarrow a = \mathsf{quotient}(a, b) \cdot b + \mathsf{remainder}(a, b)$$
$$\wedge\ (\forall a, b \in T)\, b \neq 0 \Rightarrow w(\mathsf{remainder}(a, b)) < w(b)$$

这里的 w 称为**欧几里得函数** (Euclidean function).

引理 5.19 在欧几里得半环里, $a \cdot b = 0 \Rightarrow a = 0 \vee b = 0$.

```
template<typename T>
    requires(EuclideanSemiring(T))
T gcd(T a, T b)
{
    // 前条件: ¬(a = 0 ∧ b = 0)
    while (true) {
        if (b == T(0)) return a;
        a = remainder(a, b);
        if (a == T(0)) return b;
        b = remainder(b, a);
    }
}
```

可以看出, 这里不是必须用 remainder_nonnegative, 也可以用相应类型里定义的 remainder 函数. 每次应用 remainder 总使 w 减小, 这一事实保证了终止性.

引理 5.20 gcd 在欧几里得半环里终止.

在欧几里得半环里, quotient 总返回半环里的一个元素. 这就排除了使用欧几里得原框架的条件: 任意两个可公度的量都可确定一个公共的度量. 例如, $\gcd(\frac{1}{2}, \frac{3}{4}) = \frac{1}{4}$. 可以用**欧几里得半模** (Euclidean semimodule) 的概念统一原框架和新框架, 欧几里得半模允许 quotient 返回其他类型的值, 而把 gcd 的终止性作为公理:

$$\begin{aligned}
&\mathit{EuclideanSemimodule}(\mathsf{T},\mathsf{S}) \triangleq \\
&\qquad \mathit{Semimodule}(\mathsf{T},\mathsf{S}) \\
&\quad \wedge\ \mathsf{remainder} : \mathsf{T}\times\mathsf{T}\to\mathsf{T} \\
&\quad \wedge\ \mathsf{quotient} : \mathsf{T}\times\mathsf{T}\to\mathsf{S} \\
&\quad \wedge\ (\forall a, b \in \mathsf{T})\, b \neq 0 \Rightarrow a = \mathsf{quotient}(a,b)\cdot b + \mathsf{remainder}(a,b) \\
&\quad \wedge\ (\forall a, b \in \mathsf{T})\, (a \neq 0 \vee b \neq 0) \Rightarrow \mathsf{gcd}(a,b) \text{ 终止}
\end{aligned}$$

其中 gcd 定义为

```
template<typename T, typename S>
    requires(EuclideanSemimodule(T, S))
T gcd(T a, T b)
{
    // 前条件: ¬(a = 0 ∧ b = 0)
    while (true) {
        if (b == T(0)) return a;
        a = remainder(a, b);
        if (a == T(0)) return b;
        b = remainder(b, a);
    }
}
```

由于每个可交换的半环都是定义在它自己上的半模, 只要 quotient 返回同一个类型, 这个算法就可以用, 例如对于实数上的多项式.

5.6 Stein gcd

1961 年 Josef Stein 发明了一个新的整数 gcd 算法, 该算法经常比欧几里得算法快一些 [Stein 1967]. 这个算法依赖于下面两个性质:

$$\begin{aligned}
\gcd(a,b) &= \gcd(b,a) \\
\gcd(a,a) &= a
\end{aligned}$$

还有对所有 $a > b > 0$ 的如下性质:

$$\begin{aligned}\gcd(2a, 2b) &= 2\gcd(a, b)\\ \gcd(2a, 2b+1) &= \gcd(a, 2b+1)\\ \gcd(2a+1, 2b) &= \gcd(2a+1, b)\\ \gcd(2a+1, 2b+1) &= \gcd(2b+1, a-b)\end{aligned}$$

练习 5.3 请实现整数的 Stein gcd 算法, 并证明其终止性.

看起来 Stein gcd 似乎依赖于整数的二进制表示, 但 2 是最小素整数的事实使该算法可以推广到其他的域, 其中使用该域的最小素元素. 例如, 对于多项式, 可以用 x 的一个单项式,[9] 对高斯整数, 用 $1+i$.[10] Stein gcd 算法也可以用于非欧几里得环.[11]

项目 5.2 请找出保证 Stein gcd 的既正确又具普遍性的框架.

5.7 商

对快速求商和余数的推导, 与前面有关快速余数的推导正好是平行的, 现在要从 a 除以 b 推导出一个有关商 m 和余数 u 的表达式, 用 a 除以 $2b$ 的商 n 和余数 v 表示:

$$a = n(2b) + v$$

由于余数 v 必然小于除数 $2b$, 就有

$$u = \begin{cases} v & \text{if } v < b \\ v - b & \text{if } v \geqslant b \end{cases}$$

以及

$$m = \begin{cases} 2n & \text{if } v < b \\ 2n+1 & \text{if } v \geqslant b \end{cases}$$

9. 见 Knuth [1997, 练习 4.6.1.6 (435 页) 及其解 (673 页)].

10. 见 Weilert [2000].

11. 见 Agarwal and Frandsen [2004].

这样就得到了下面的代码:

```
template<typename T>
    requires(ArchimedeanMonoid(T))
pair<QuotientType(T), T>
quotient_remainder_nonnegative(T a, T b)
{
    // 前条件: a ≥ 0 ∧ b > 0
    typedef QuotientType(T) N;
    if (a < b) return pair<N, T>(N(0), a);
    if (a - b < b) return pair<N, T>(N(1), a - b);
    pair<N, T> q = quotient_remainder_nonnegative(a, b + b);
    N m = twice(q.m0);
    a = q.m1;
    if (a < b) return pair<N, T>(m, a);
    else       return pair<N, T>(successor(m), a - b);
}
```

如果有“折半” (halving) 操作, 就可以得到下面代码:

```
template<typename T>
    requires(HalvableMonoid(T))
pair<QuotientType(T), T>
quotient_remainder_nonnegative_iterative(T a, T b)
{
    // 前条件: a ≥ 0 ∧ b > 0
    typedef QuotientType(T) N;
    if (a < b) return pair<N, T>(N(0), a);
    T c = largest_doubling(a, b);
    a = a - c;
    N n(1);
    while (c != b) {
        n = twice(n);
```

```
        c = half(c);
        if (c <= a) {
            a = a - c;
            n = successor(n);
        }
    }
    return pair<N, T>(n, a);
}
```

5.8 负量的商和余数

许多计算机教授和程序设计语言对负量的商和余数的定义都是不对的. 作为有关阿基米德幺半群的定义的扩展, *阿基米德群* (Euclidean group) T 必须满足下面性质, 其中的 $b \neq 0$:

$$a = \mathsf{quotient}(a, b) \cdot b + \mathsf{remainder}(a, b)$$

$$|\mathsf{remainder}(a, b)| < |b|$$

$$\mathsf{remainder}(a + b, b) = \mathsf{remainder}(a - b, b) = \mathsf{remainder}(a, b)$$

最后一个性质等价于经典数学里*同余* (congruence) 的定义.[12] 有关数论的书籍里通常假定 $b > 0$, 可以将 remainder 相容地扩展到 $b < 0$ 的情况. 如果求余的实现将商向零的方向截取, 就会违背上面的第 3 条, 从而不满足我们的需求.[13] 除了违背上面的第 3 条要求外, 截取还是一种低劣的约简方法, 因为它送出 0 的次数是送出任何其他整数的次数的两倍, 这导致了一种不一致的分布.

要给出满足上面对非负输入的三条要求的求余数过程 rem 和求商过程 quo_rem, 可以写两个适配器过程, 使之对于正的和负的整数都给出正确结果. 这些适配器过程可以用在*阿基米德群*上:

---

12. "如果两个数 a 和 b 相对于同一模数 k 有同样的余数 r, 则称它们相对于模数 k 同余 (源自高斯)" [Dirichlet 1863].

13. 关于商和余数的卓越讨论参见 Boute [1992]. Boute 指出了两个可以接受的扩充, 分别称为 E 和 F. 我们将采用 Knuth 更喜欢的被 Boute 称为 F 的扩充.

ArchimedeanGroup(T) ≜

ArchimedeanMonoid(T)

∧ *AdditiveGroup*(T)

```
template<typename Op>
    requires(BinaryOperation(Op) && ArchimedeanGroup(Domain(Op)))
Domain(Op) remainder(Domain(Op) a, Domain(Op) b, Op rem)
{
    // 前条件: b ≠ 0
    typedef Domain(Op) T;
    T r;
    if (a < T(0))
        if (b < T(0)) {
            r = -rem(-a, -b);
        } else {
            r =  rem(-a,  b); if (r != T(0)) r = b - r;
        }
    else
        if (b < T(0)) {
            r =  rem(a, -b);  if (r != T(0)) r = b + r;
        } else {
            r =  rem(a,  b);
        }
    return r;
}

template<typename F>
    requires(HomogeneousFunction(F) && Arity(F) == 2 &&
        ArchimedeanGroup(Domain(F)) &&
        Codomain(F) == pair<QuotientType(Domain(F)),
                            Domain(F)>)
pair<QuotientType(Domain(F)), Domain(F)>
```

```
quotient_remainder(Domain(F) a, Domain(F) b, F quo_rem)
{
    // 前条件: b ≠ 0
    typedef Domain(F) T;
    pair<QuotientType(T), T> q_r;
    if (a < T(0)) {
        if (b < T(0)) {
            q_r = quo_rem(-a, -b); q_r.m1 = -q_r.m1;
        } else {
            q_r = quo_rem(-a,  b);
            if (q_r.m1 != T(0)) {
                q_r.m1 = b - q_r.m1; q_r.m0 = successor(q_r.m0);
            }
            q_r.m0 = -q_r.m0;
        }
    } else {
        if (b < T(0)) {
            q_r = quo_rem( a, -b);
            if (q_r.m1 != T(0)) {
                q_r.m1 = b + q_r.m1; q_r.m0 = successor(q_r.m0);
            }
            q_r.m0 = -q_r.m0;
        }
        else
            q_r = quo_rem( a,  b);
    }
    return q_r;
}
```

引理 5.21 只要 remainder 和 quotient_remainder 的函数参数满足对于正参数的要求, 这两个过程就能满足我们的要求.

5.9 概念及其模型

从第 2 章起我们一直使用整数类型, 但是并没有给出这个概念的形式化定义. 本章前面部分定义了有序代数结构, 在此基础上已经可以形式化地处理整数了. 首先定义**离散阿基米德半环** (discrete Archimedean semiring):

$$
\begin{aligned}
&\mathit{DiscreteArchimedeanSemiring}(\mathsf{T}) \triangleq \\
&\quad \mathit{CommutativeSemiring}(\mathsf{T}) \\
&\wedge\ \mathit{ArchimedeanMonoid}(\mathsf{T}) \\
&\wedge\ (\forall a, b, c \in \mathsf{T})\, a < b \wedge 0 < c \Rightarrow a \cdot c < b \cdot c \\
&\wedge\ \neg(\exists a \in \mathsf{T})\, 0 < a < 1
\end{aligned}
$$

离散性 (discreteness) 就是指上面的最后一条性质, 即, 在 0 和 1 之间不存在任何元素.

离散的阿基米德半环可以有负元素. 与负元素不存在相关的概念是

$$
\begin{aligned}
&\mathit{NonnegativeDiscreteArchimedeanSemiring}(\mathsf{T}) \triangleq \\
&\quad \mathit{DiscreteArchimedeanSemiring}(\mathsf{T}) \\
&\wedge\ (\forall a \in \mathsf{T})\, 0 \leqslant a
\end{aligned}
$$

离散阿基米德半环不存在加法的逆; 与加法逆相关的概念是

$$
\begin{aligned}
&\mathit{DiscreteArchimedeanRing}(\mathsf{T}) \triangleq \\
&\quad \mathit{DiscreteArchimedeanSemiring}(\mathsf{T}) \\
&\wedge\ \mathit{AdditiveGroup}(\mathsf{T})
\end{aligned}
$$

两个类型 T 和 T′ **同构** (isomorphic), 如果可能写出从 T 到 T′ 以及从 T′ 到 T 的转换函数, 它们能保持两个类型的过程及公理.

一个概念是**单叶的** (univalent), 如果任何满足它的概念都相互同构. 概念 *NonnegativeDiscreteArchimedeanSemiring* 是单叶的; 满足它的类型都与自然数类型 $\mathbb{N}$ 同构.[14] *DiscreteArchimedeanRing* 也是单叶的; 满足它的类型都与整数类型 $\mathbb{Z}$ 同构. 正如在上面看到的, 增加新公理将导致满足概念的模型的减少, 使得一个概念很快达到一个单叶点.

14. 我们按照 Peano [1908, 27 页] 的定义将 0 包含在自然数中.

本章将继续以演绎的方式工作, 通过增加运算和公理, 从比较一般的概念演绎出更特殊的概念. 这一推导方法能静态地展示一个概念以及与其相关定理和算法的分类体系. 实际的发现过程则按归纳的方式进行, 从如整数或实数一类的具体模型开始, 去掉一些运算和公理, 设法找到某些有趣的算法还能继续应用的最弱概念.

在定义一个概念时, 必须验证其公理的独立性和协调性, 并阐明其有用性.

一个命题**独立于** (independence of) 一集公理, 如果存在一个模型使这些公理都为真, 但是这一命题却为假. 例如, 可结合性与可交换性相互独立: 串拼接可结合但不可交换, 而两个值的平均值 ($\frac{x+y}{2}$) 可交换但不可结合. 一个命题**依赖于** (dependent) 一集公理, 如果它可以由这些公理导出.

一个概念是**协调的** (consistency), 如果它存在模型. 继续我们的例子, 自然数的加法是可交换且可结合的. 一个概念是**不协调的** (inconsistency), 如果有某个命题和它的否定都可以由这个概念的公理集合导出. 换句话说, 要阐释一个概念的协调性, 我们只需要构造出它的一个模型; 要阐释其不协调性, 只需要推导出一个矛盾.

一个概念是**有用的** (usefulness), 如果存在某个有用的算法, 使得这一概念是该算法的最抽象的场景. 举例说, 并行的不考虑顺序的归约适用于任何可结合且可交换的运算.

5.10 计算机整数类型

计算机指令集通常提供自然数和整数的某种部分表示. 例如可能有一个**有界无符号二进制整数类型** (bounded unsigned binary integer type) U_n, 其中的 $n = 8, 16, 32, 64, \ldots$, 它可以表示区间 $[0, 2^n)$ 里的任何值. 另有一个**有界的带符号整数类型** (bounded integer type) S_n, $n = 8, 16, 32, 64, \ldots$, 它能表示区间 $[-2^{n-1}, 2^{n-1})$ 里的任何值. 虽然这些类型是有界的, 典型的计算机指令实现的却是它们上面的全运算, 将结果都编码为有界值的二元组.

对于有界无符号类型, 通常存在具有下面签名的指令:

$$\text{sum\_extended} : U_n \times U_n \times U_1 \to U_1 \times U_n$$

$$\text{difference\_extended} : U_n \times U_n \times U_1 \to U_1 \times U_n$$

$$\text{product\_extended} : U_n \times U_n \to U_{2n}$$

$$\text{quotient\_remainder\_extended} : U_n \times U_n \to U_n \times U_n$$

注意, U_{2n} 可以表示为 $U_n \times U_n$ (一对 U_n). 如果程序语言提供了对这些硬件运算的完全访问, 就可能用它写出涉及整数类型的高效的抽象的软件部件.

项目 5.3 请为有界的无符号和带符号二进制整数设计一组概念. 有关现代计算机体系结构的研究已经弄清了它应该包含的功能. MMIX 是这些指令集合的一个很好的抽象 [Knuth 2005].

5.11 结论

可以将各种算法和数学结构无缝地组合为一个整体, 方法是用抽象的术语来描述算法, 并根据算法的需要去调整理论. 本章的数学和算法是一些已有两千多年历史的结果的重新叙述.

第 6 章

迭代器

本章介绍**迭代器**的概念, 它是算法和顺序数据结构之间的接口. 迭代器概念的一种分层结构说明了不同类型的顺序遍历: 一遍前向的、多遍前向的、双向的, 以及随机访问.[1] 我们将考察一些常见算法 (如线性和二分检索) 的各种接口. 有界和计数范围是为各种顺序算法定义接口的很灵活的机制.

6.1 可读性

每个对象有一个地址, 即, 到计算机内存的一个整数索引. 程序通过地址访问和修改对象. 此外, 地址还使程序可以创建各种各样的数据结构, 其中许多结构依赖于一个事实: 地址就是整数, 可以应用与整数类似的运算.

迭代器 (iterator) 是一族概念, 它们抽象了地址的不同方面, 使写出的算法不仅可以对地址工作, 而且能对任何满足最小的一集需求的类似地址的对象工作. 第 7 章将介绍一个更广泛的概念族: **坐标结构** (coordinate structures).

迭代器上有两类运算: 访问值和遍历. 存在三种访问: 读、写, 以及读和写. 存在四种线性遍历: 一遍前向的 (输入流)、多遍前向的 (单链表)、双向的 (双链表) 和随机访问 (数组).

本章研究第一种访问: 可读性, 也就是说, 获得被另一个对象指称的对象的值的能力. 一个类型 T 是**可读的** (readable), 如果其上定义了一个一元函数 source, 该函数返回一个类型为 ValueType(T) 的对象:

Readable(T) ≜

Regular(T)

---

1. 这里有关迭代器的处理是 Stepanov and Lee [1995] 工作的精化, 但又在几个方面与之不同.

$\wedge$ ValueType : $Readable \rightarrow Regular$

$\wedge$ source : T $\rightarrow$ ValueType(T)

source 只在需要值的上下文中使用, 其结果可作为值或者常量引用传给过程.

可读类型里也可能存在使 source 无定义的对象; 也就是说 source 不必是全的. 相关概念不提供定义空间谓词来帮助确定 source 是否对某特定对象有定义. 举例说, 给了一个指向类型 T 的指针, 不可能确定它是否指着一个具有合法结构的对象. 算法里使用函数 source 的合法性只能通过前条件推导出来.

对某可读类型的一个对象, 以调用 source 的方式访问之, 这一访问在效率上将不劣于通过其他任何方式访问同一数据. 特别的, 对位于内存的一个可读的具有值类型 T 的对象, 我们期望执行 source 的代价近似等于对一个到 T 的常规指针做间接访问的代价. 当然, 就像常规指针一样, 这里也可能出现由于存储器结构而带来的代价不统一的情况. 总而言之, 上面说法也就是想说明, 完全没有必要为加速一个算法而考虑用指针来代替迭代器.

扩展 source 的定义, 使之可以用于那些其对象并不指向其他对象的类型, 这种做法也可能很有用. 这里的做法是, 在 source 作用于这种对象时, 就让它直接返回自己的参数. 这一假设使我们可以在程序里以统一的方式描述对可读对象的值需求, 无论是对类型 T 本身, 或是对指向类型 T 的指针, 或是更一般的, 对任何值类型为 T 的对象. 因此, 除非另有定义, 我们总假定有 ValueType(T) = T, 而此时 source 返回它所作用的对象.

6.2 迭代器

要做遍历, 就需要有生成新迭代器的功能. 正如在第 2 章里已经看到的, 变换是一种生成某类型的新值的方法. 而如果一个变换是规范的, 某些一遍算法就可以不要求遍历的规范性. 进一步说, 确实有一些模型不能提供规范遍历, 例如输入流. 这样, 最弱的迭代器概念只要求伪变换 (pseudotrasformation)[2] successor 和类型函数 DistanceType:

$Iterator$(T) $\triangleq$

$Regular$(T)

2. 伪变换具有与变换一样的签名, 但它们不是规范的.

∧ DistanceType : *Iterator* → *Integer*

∧ successor : T → T

∧ successor 不必是规范的

DistanceType 返回一个整数类型, 其规模足以度量相应的迭代器类型所允许的 successor 应用的任意序列的长度. 由于本书默认地假定了规范性, 所以现在必须明确说明, 对 successor 并没有规范性的要求.

与作用于可读类型的 source 一样, successor 也不必是全的. 在迭代器类型里可能存在使得 successor 无定义的对象. 有关的概念并不为 successor 提供可用于确定它是否对某个特定对象有定义的定义空间谓词. 举例说, 指向一个数组的指针并不包含信息来说明对它可以做多少次增量. 在一个算法里使用 successor 的合法性必须由前条件推导出来.

下面代码定义了与 successor 对应的动作:

```
template<typename I>
    requires(Iterator(I))
void increment(I& x)
{
    // 前条件: successor(x) 有定义
    x = successor(x);
}
```

许多重要算法是一遍的 (single-pass), 如线性检索和复制; 也就是说, 它们仅将 successor 应用到每个迭代器的值上一次. 这就使它们可以作用于输入流, 这也是我们丢掉对 successor 的规范性要求的原因: 即使 successor 都有定义, i = j 也未必蕴涵 successor(i) = successor(j). 进一步说, 在调用 successor(i) 之后, i 和任何等于它的迭代器都不再是良形式的了. 但它们仍然是部分有效的, 可以销毁它们或向它们赋值; 但不能再对它们使用 successor、source 和 =.

注意, successor(i) = successor(j) 并不蕴涵着 i = j. 请考虑 (例如) 两个用空指针结尾的单链表.

迭代器提供了一种遍历整个数据集合的方法, 而且其效率相当于在同样数据上遍历的任何其他方法.

为使一个整数类型能建模 *Iterator*, 它必须有一个距离类型. 无符号整数类型是它自己的距离类型; 对于任何有界二进制整数类型 $\mathtt{S}_n$, 其距离类型是与之对应的无符号类型 $\mathtt{U}_n$.

6.3 范围

设 f 是某个迭代器类型的一个对象, 而 n 是与之对应的距离类型的一个对象, 我们希望能定义在从 f 开始的 n 个迭代器的**弱范围** (weak range) $[\![\mathtt{f}, \mathtt{n})\!)$ 上的算法, 采用下面形式的代码

```
while (!zero(n)) { n = predecessor(n); ... f = successor(f); }
```

下面性质使我们可以做这种迭代:

property(I : *Iterator*)
weak_range : I × DistanceType(I)
$(\mathtt{f}, \mathtt{n}) \mapsto (\forall \mathtt{i} \in \mathsf{DistanceType}(\mathtt{I}))$
$(0 \leqslant \mathtt{i} \leqslant \mathtt{n}) \Rightarrow \mathsf{successor}^{\mathtt{i}}(\mathtt{f})$ 有定义

引理 6.1 $0 \leqslant \mathtt{j} \leqslant \mathtt{i} \wedge \mathsf{weak\_range}(\mathtt{f}, \mathtt{i}) \Rightarrow \mathsf{weak\_range}(\mathtt{f}, \mathtt{j})$

在一个弱范围里可以前进的步数由它的规模确定:

```
template<typename I>
    requires(Iterator(I))
I operator+(I f, DistanceType(I) n)
{
    // 前条件: n ≥ 0 ∧ weak_range(f, n)
    while (!zero(n)) {
        n = predecessor(n);
        f = successor(f);
    }
    return f;
}
```

加上下面公理就能保证在此范围里不存在环路:

property(I : *Iterator*, N : *Integer*)

counted_range : I × N

$(f, n) \mapsto \text{weak\_range}(f, n) \wedge$

$(\forall i, j \in N)\,(0 \leqslant i < j \leqslant n) \Rightarrow$

$\text{successor}^i(f) \neq \text{successor}^j(f)$

当 f 和 l 是同一迭代器类型的对象时, 我们希望采用下面形式的代码

```
while (f != l) { ... f = successor(f); }
```

定义在从 f 开始由 l 界定的有界范围 (bounded range) [f, l) 上的算法.

有界性质使人可以写出这种迭代:

property(I : *Iterator*)

bounded_range : I × I

$(f, l) \mapsto (\exists k \in \text{DistanceType}(I))\ \text{counted\_range}(f, k) \wedge \text{successor}^k(f) = l$

这种结构的迭代使用了一个有界范围, 到第一次遇到 l 时结束. 这样, 有界范围就与弱范围不一样, 这里不允许有环 (cycle).

对一个有界范围, 可以实现迭代上的部分减法:[3]

```
template<typename I>
    requires(Iterator(I))
DistanceType(I) operator-(I l, I f)
{
    // 前条件: bounded_range(f,l)
    DistanceType(I) n(0);
    while (f != l) {
        n = successor(n);
        f = successor(f);
    }
    return n;
}
```

3. 请注意与第 2 章的 distance 的类似之处.

由于 successor 可能不是规范的, 因此只能在一定的前条件下, 或在那些只是要计算有界范围规模的情景中使用减法.

迭代器和整数之间的 + 和 − 的定义与数学里的用法不同, 因为数学里的 + 和 − 只定义在同类型的对象之间. 另一方面, 迭代器和整数之间的 +, 以及迭代器之间的 − 都是基于 successor 归纳定义的, 这些又与数学里一样. 自然数上加法的标准归纳定义基于函数 successor:[4]

$$a + 0 = a$$

$$a + \mathrm{successor}(b) = \mathrm{successor}(a + b)$$

对迭代器的 $f + n$ 的归纳定义与此等价, 但所涉及的 f 和 n 具有不同类型. 一种变形的结合性也可以通过归纳法证明, 这也与自然数里的情况类似.

引理 6.2 $(f + n) + m = f + (n + m)$

在前条件里需要描述范围的成员关系. 我们借用描述区间的记法约定 (见附录 A), 引进*半开的*范围和*闭的*范围. 下面用几种稍微不同的写法分别表示弱范围、计数范围和有界范围.

一个半开的弱范围或计数范围 (half-open weak range, half-open counted range)$[\![f, n)\!)$, 其中 $n \geqslant 0$ 是整数, 表示的是迭代器序列 $\{\mathrm{successor}^k(f) \mid 0 \leqslant k < n\}$. *一个闭的弱范围或计数范围* (closed weak range, closed counted range) $[\![f, n]\!]$, 其中 $n \geqslant 0$ 是整数, 表示的是迭代器序列 $\{\mathrm{successor}^k(f) \mid 0 \leqslant k \leqslant n\}$.

半开有界范围 $[f, l)$ 等价于半开计数范围 $[\![f, l - f)\!)$. 闭有界范围 $[f, l]$ 等价于闭计数范围 $[\![f, l - f]\!]$.

一个范围的*规模* (大小) 就是它表示的序列里的迭代器个数.

引理 6.3 对于半开范围里的每个迭代器, 以及闭范围里除最后一个之外的每个迭代器, successor 都有定义.

如果 r 是范围而 i 是迭代器, $i \in r$ 表示 i 是 r 的迭代器集合的成员.

引理 6.4 如果 $i \in [f, l)$, 那么 $[f, i)$ 和 $[i, l)$ 都是有界范围.

空的半开范围 (empty half-open range) 用 $[\![i, 0)\!)$ 或 $[i, i)$ 描述, 其中 i 是某个迭代器. 不存在空的闭范围.

---

4. 最早出现在 Grassmann [1861]; Grassmann 的定义由于 Peano [1908] 而广为人知.

引理 6.5 $i \notin [\![i, 0)\!) \wedge i \notin [i, i)$

引理 6.6 空范围没有首元素, 也没有末元素.

有时也需要描述从某个特定迭代器开始的空序列. 例如, 需要用二分检索找到其值都等于某个给定值的迭代器序列. 如果不存在这种值, 得到的序列就是空的, 但它却有一个位置, 如果要做插入就应该放在这里.

迭代器 l 称为半开有界范围 [f, l) 的极限 (limit in a range). 迭代器 f + n 是半开弱范围 $[\![f, n)\!)$ 的极限. 注意, 空范围也有极限, 虽然它没有首元素和末元素.

引理 6.7 半开弱范围 $[\![f, n)\!)$ 的规模是 n. 闭弱范围 $[\![f, n]\!]$ 的规模是 n + 1. 半开有界范围 [f, l) 的规模是 l − f. 闭有界范围 [f, l] 的规模是 (l − f) + 1.

如果 i 和 j 是一个计数范围或有界范围里的迭代器, 定义关系 $i \prec j$ 表示 $i \neq j \wedge \mathsf{bounded\_range}(i, j)$. 也就是说, 通过应用一次或几次 successor 就能从 i 得到 j. 关系 $\prec$ ("先于", precede) 以及与之对应的自反关系 $\preceq$ ("先于或等于", precedes or equal) 都可以用在规程里, 例如写在算法的前条件和后条件里. $\prec$ 对一个迭代器类型的很多对值都没有定义, 因此通常不存在写出实现 $\prec$ 的代码的有效方法. 举例说, 不存在有效方法来确定在一个链接结构里某个结点是否先于另一个; 两个结点也可能根本就没有链接在一起.

6.4 可读范围

对一个建模 *Readable* 和 *Iterator* 的类型, 属于该类型的一个迭代器范围称为可读的 (readable), 如果 source 对该范围里的所有迭代器都有定义:

property(I : *Readable*)
  requires(Iterator(I))
readable_bounded_range : I × I
  (f, l) ↦ bounded_range(f, l) ∧ (∀i ∈ [f, l)) source(i) 有定义

请注意, source 不必对这一范围的极限有定义. 还有, 由于在应用了 successor 之后不能保证迭代器仍然是良好的, 因此, 在取得了一个迭代器的后继之后, 不再保证还能将 source 作用于它. 可以类似地定义 readable_weak_range 和 readable_counted_range .

对一个可读范围, 可以将一个过程应用于该范围里的每个值:

```
template<typename I, typename Proc>
    requires(Readable(I) && Iterator(I) &&
        Procedure(Proc) && Arity(Proc) == 1 &&
        ValueType(I) == InputType(Proc, 0))
Proc for_each(I f, I l, Proc proc)
{
    // 前条件: readable_bounded_range(f, l)
    while (f != l) {
        proc(source(f));
        f = successor(f);
    }
    return proc;
}
```

这里返回使用的过程, 因为它可能已经在遍历中积累了一些有用信息.[5]

下面过程实现线性检索:

```
template<typename I>
    requires(Readable(I) && Iterator(I))
I find(I f, I l, const ValueType(I)& x)
{
    // 前条件: readable_bounded_range(f, l)
    while (f != l && source(f) != x) f = successor(f);
    return f;
}
```

这个过程结束时, 或者是它返回的迭代器等于范围的极限, 或者该迭代器的值等于 x. 返回极限说明所做的检索失败. 由于对规模为 n 的范围的检索有 $n+1$ 种可能结果, 极限值在这里很有用, 许多其他算法里也有这种情况. 如果还需要用 find 做检索, 只需对返回的迭代器之后的下一个位置再次调用它.

修改与 x 的比较, 用相等而不是不等做判断, 就能得到 find_not.

5. 可以在这里使用函数对象.

可以把检索某个值推广到检索满足某个一元谓词的第一个值:

```
template<typename I, typename P>
    requires(Readable(I) && Iterator(I) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
I find_if(I f, I l, P p)
{
    // 前条件: readable_bounded_range(f, l)
    while (f != l && !p(source(f))) f = successor(f);
    return f;
}
```

使用谓词本身而不是该谓词的补, 得到的就是 find_if_not.

练习 6.1 请用 find_if 和 find_if_not 实现量词函数 all、none、not_all 和 some, 它们都以一个有界范围和一个谓词作为参数.

结合使用算法 find 和量词函数, 可以检索满足某条件的一些值. 还可以统计这种值的个数:

```
template<typename I, typename P, typename J>
    requires(Readable(I) && Iterator(I) &&
        UnaryPredicate(P) && Iterator(J) &&
        ValueType(I) == Domain(P))
J count_if(I f, I l, P p, J j)
{
    // 前条件: readable_bounded_range(f, l)
    while (f != l) {
        if (p(source(f))) j = successor(j);
        f = successor(f);
    }
    return j;
}
```

如果给 j 加一个整数需要线性时间, 那么就应该明确地把 j 作为一个参数. 类型 J 可以是任何整数或迭代器类型, 包括 I.

练习 6.2 请换一种方式实现 count_if, 其中把一个适当的函数对象送给 for_each, 最后从返回的函数对象里取得作为结果的值.

最自然的默认方法是从 0 开始计数, 并使用迭代器的距离类型:

```
template<typename I, typename P>
    requires(Readable(I) && Iterator(I) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
DistanceType(I) count_if(I f, I l, P p) {
    // 前条件: readable_bounded_range(f,l)
    return count_if(f, l, p, DistanceType(I)(0));
}
```

把其中的谓词换成相等检查, 就能得到 count; 把检查反过来就可以得到 count_not 和 count_if_not.

对 a_i 的求和记为 $\sum_{i=0}^{n} a_i$, 这种记法可以推广到很多二元运算. 例如, $\prod_{i=0}^{n} a_i$ 表示乘积, 而 $\bigwedge_{i=0}^{n} a_i$ 表示合取. 这些例子里用的运算都是可结合的, 这就意味着分组并不重要. Kenneth Iverson 在程序设计语言 APL 里把这些写法统一到一个**归约运算符** (reduction operator) /, 它以一个二元运算和一个序列为参数, 从序列的元素归约出最后的结果.[6] 例如, +/1 2 3 等于 6.

Iverson 并没有把归约限制到可结合运算. 可以扩展 Iverson 的归约, 使之能在迭代器范围上工作, 但只限制到**部分可结合** (partially associative) 运算. 部分可结合是指, 如果该运算在相邻元素上都有定义, 它就能任意结合:

property(Op : *BinaryOperation*)
partially_associative : Op
    op $\mapsto$ ($\forall$a, b, c $\in$ Domain(op))
        如果 op(a, b) 和 op(b, c) 有定义,

6. 见 Iverson [1962].

那么 $\mathrm{op}(\mathrm{op}(a,b),c)$ 和 $\mathrm{op}(a,\mathrm{op}(b,c))$ 也都有定义且相等.

作为部分可结合但却不是可结合运算的例子, 请考虑两个范围 $[f_0,l_0)$ 和 $[f_1,l_1)$ 的拼接, 显然只有 $l_0=f_1$ 时它才有定义.

这里允许在执行二元运算前将一个一元函数作用于每个迭代器, 由 i 得到 a_i. 因为一个任意的部分可结合运算可能没有单位元, 这里先给出一个归约的定义, 它要求参数是非空范围:

```
template<typename I, typename Op, typename F>
    requires(Iterator(I) && BinaryOperation(Op) &&
        UnaryFunction(F) &&
        I == Domain(F) && Codomain(F) == Domain(Op))
Domain(Op) reduce_nonempty(I f, I l, Op op, F fun)
{
    // 前条件: bounded_range(f,l) ∧ f ≠ l
    // 前条件: partially_associative(op)
    // 前条件: (∀x ∈ [f,l)) fun(x) 有定义
    Domain(Op) r = fun(f);
    f = successor(f);
    while (f != l) {
        r = op(r, fun(f));
        f = successor(f);
    }
    return r;
}
```

fun 最自然的默认值是 source. 可以给过程传一个单位元素, 用它作为空范围的返回值:

```
template<typename I, typename Op, typename F>
    requires(Iterator(I) && BinaryOperation(Op) &&
        UnaryFunction(F) &&
        I == Domain(F) && Codomain(F) == Domain(Op))
Domain(Op) reduce(I f, I l, Op op, F fun, const Domain(Op)& z)
```

```
{
    // 前条件: bounded_range(f, l)
    // 前条件: partially_associative(op)
    // 前条件: (∀x ∈ [f, l)) fun(x) 有定义
    if (f == l) return z;
    return reduce_nonempty(f, l, op, fun);
}
```

如果涉及单位元的运算比较慢, 或者是这样的操作要求实现另一种求解逻辑, 下面这个过程可能很有用:

```
template<typename I, typename Op, typename F>
    requires(Iterator(I) && BinaryOperation(Op) &&
        UnaryFunction(F) &&
        I == Domain(F) && Codomain(F) == Domain(Op))
Domain(Op) reduce_nonzeroes(I f, I l,
                            Op op, F fun, const Domain(Op)& z)
{
    // 前条件: bounded_range(f, l)
    // 前条件: partially_associative(op)
    // 前条件: (∀x ∈ [f, l)) fun(x) 有定义
    Domain(Op) x;
    do {
        if (f == l) return z;
        x = fun(f);
        f = successor(f);
    } while (x == z);
    while (f != l) {
        Domain(Op) y = fun(f);
        if (y != z) x = op(x, y);
        f = successor(f);
    }
```

```
    return x;
}
```

对任何以有界范围为参数的算法, 都存在与之对应的以弱范围或计数范围为参数的版本; 但它们需要返回更多信息:

```
template<typename I, typename Proc>
    requires(Readable(I) && Iterator(I) &&
        Procedure(Proc) && Arity(Proc) == 1 &&
        ValueType(I) == InputType(Proc, 0))
pair<Proc, I> for_each_n(I f, DistanceType(I) n, Proc proc)
{
    // 前条件: readable_weak_range(f, n)
    while (!zero(n)) {
        n = predecessor(n);
        proc(source(f));
        f = successor(f);
    }
    return pair<Proc, I>(proc, f);
}
```

这里必须返回迭代器的最终值. 因为没有要求 successor 的规范性, 这就意味着不可能简单地算出这个最终值. 即使 successor 是规范的, 重新计算迭代器也需要与范围的规模成正比的时间.

```
template<typename I>
    requires(Readable(I) && Iterator(I))
pair<I, DistanceType(I)> find_n(I f, DistanceType(I) n,
                                const ValueType(I)& x)
{
    // 前条件: readable_weak_range(f, n)
    while (!zero(n) && source(f) != x) {
        n = predecessor(n);
        f = successor(f);
```

```
    }
    return pair<I, DistanceType(I)>(f, n);
}
```

find_n 返回迭代器的最终值和计数值, 因为重新开始搜索需要用它们.

练习 6.3 请实现一种算法变形, 其中使用的 find、量词、count 和 reduce 的版本都以弱范围为参数, 而不是以受限范围为参数.

如果能保证所给范围里必定存在满足谓词的元素, 就可以删去 find_if 循环里的两个检测中的一个. 这种满足谓词的元素称为*哨兵* (sentinel):

```
template<typename I, typename P>
    requires(Readable(I) && Iterator(I) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
I find_if_unguarded(I f, P p) {
    // 前条件: (∃l) readable_bounded_range(f, l) ∧ some(f, l, p)
    while (!p(source(f))) f = successor(f);
    return f;
    // 后条件: p(source(f))
}
```

如果应用该谓词本身而不是其否定, 就得到 find_if_not_unguarded.

给定了两个具有相同值类型的范围和该值类型上的一个关系, 可以检索不匹配的值对:

```
template<typename I0, typename I1, typename R>
    requires(Readable(I0) && Iterator(I0) &&
        Readable(I1) && Iterator(I1) && Relation(R) &&
        ValueType(I0) == ValueType(I1) &&
        ValueType(I0) == Domain(R))
pair<I0, I1> find_mismatch(I0 f0, I0 l0, I1 f1, I1 l1, R r)
{
    // 前条件: readable_bounded_range(f0, l0)
    // 前条件: readable_bounded_range(f1, l1)
```

```
    while (f0 != l0 && f1 != l1 && r(source(f0), source(f1))) {
        f0 = successor(f0);
        f1 = successor(f1);
    }
    return pair<I0, I1>(f0, f1);
}
```

练习 6.4 请给出 find_mismatch 的后条件, 并解释为什么两个迭代器的最终值会是这种情况.

适合用在 find_mismatch 里的最自然的默认关系是值类型上的相等关系.

练习 6.5 请为限界范围和弱范围的四种组合设计 find_mismatch 的各种变形.

有时需要做的不是在两个范围里找到不匹配的元素, 而是在一个范围里找不匹配的相邻元素:

```
template<typename I, typename R>
    requires(Readable(I) && Iterator(I) &&
        Relation(R) && ValueType(I) == Domain(R))
I find_adjacent_mismatch(I f, I l, R r)
{
    // 前条件: readable_bounded_range(f,l)
    if (f == l) return l;
    ValueType(I) x = source(f);
    f = successor(f);
    while (f != l && r(x, source(f))) {
        x = source(f);
        f = successor(f);
    }
    return f;
}
```

如前所述, 在对一个迭代器使用了 successor 之后, 就不能再对它应用 source. 因此这里必须拷贝它以前的值. 对 *Iterator* 的弱要求也意味着, 如果找

到不匹配的对之后返回第一个迭代器, 实际返回的可能不是一个良形式的值.

6.5 递增的范围

给定了某种类型的迭代器的值类型上的一个关系, 该迭代器类型的一个范围称为是对它保关系的 (relation preserving), 如果该关系对这一范围里所有的相邻值都成立. 换句话说, 如果在这一范围和关系下调用 find_adjacent_mismatch, 返回的将是范围的极限:

```
template<typename I, typename R>
    requires(Readable(I) && Iterator(I) &&
        Relation(R) && ValueType(I) == Domain(R))
bool relation_preserving(I f, I l, R r)
{
    // 前条件: readable_bounded_range(f, l)
    return l == find_adjacent_mismatch(f, l, r);
}
```

给定了一个弱序 r, 如果一个范围对 r 的逆的补是保关系的, 那么就说这一范围是 r-递增的 (increasing range). 给定弱关系 r, 如果一个范围对于 r 是保关系的, 则称这一范围是严格 r-递增的 (strictly increasing range).[7] 很容易为严格递增范围实现一个检测:

```
template<typename I, typename R>
    requires(Readable(I) && Iterator(I) &&
        Relation(R) && ValueType(I) == Domain(R))
bool strictly_increasing_range(I f, I l, R r)
{
    // 前条件: readable_bounded_range(f, l) ∧ weak_ordering(r)
    return relation_preserving(f, l, r);
}
```

可以借助于函数对象实现一个递增范围的检测:

7. 也有些作者使用术语非递减和递增, 而不是递增和严格递增.

```
template<typename R>
    requires(Relation(R))
struct complement_of_converse
{
    typedef Domain(R) T;
    R r;
    complement_of_converse(const R& r) : r(r) { }
    bool operator()(const T& a, const T& b)
    {
        return !r(b, a);
    }
};

template<typename I, typename R>
    requires(Readable(I) && Iterator(I) &&
        Relation(R) && ValueType(I) == Domain(R))
bool increasing_range(I f, I l, R r)
{
    // 前条件: readable_bounded_range(f, l) ∧ weak_ordering(r)
    return relation_preserving(
        f, l,
        complement_of_converse<R>(r));
}
```

strictly_increasing_counted_range 和 increasing_counted_range 的定义直截了当.

给定某迭代器类型的值类型上的谓词 p, 如果在该迭代器类型的一个范围里, 所有满足该谓词的值都在此范围里所有不满足该谓词的值之后, 就称这一范围是 p-划分的 (p-partitioned range). 很容易写出一个能检查作为参数的范围是否 p-划分的过程:

```
template<typename I, typename P>
    requires(Readable(I) && Iterator(I) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
```

```
bool partitioned(I f, I l, P p)
{
    // 前条件: readable_bounded_range(f,l)
    return l == find_if_not(find_if(f, l, p), l, p);
}
```

调用 find_if 返回的迭代器称为划分点 (partition point). 它是相关的值能满足这一谓词的第一个迭代器 (如果存在).

练习 6.6 请实现谓词 partitioned_n, 它检测一个计数范围是否为 p-划分的.

线性检索需要在每次应用 successor 之后调用 source, 因为失败的检测不能为这一范围里的任何迭代器的值提供信息. 当然, 划分范围的规范性给了我们更多的信息.

引理 6.8 如果 p 是一个谓词, $[f,l)$ 是一个 p-划分的范围, 那么:

$$(\forall m \in [f,l))\,\neg p(\mathsf{source}(m)) \Rightarrow (\forall j \in [f,m])\,\neg p(\mathsf{source}(j))$$
$$(\forall m \in [f,l))\,p(\mathsf{source}(m)) \Rightarrow (\forall j \in [m,l))\,p(\mathsf{source}(j))$$

这说明了一个找到划分点的二分段算法: 如果数据是一致分布的, 检测范围的中点能把检索空间缩小一半. 当然, 这个算法可能需要去遍历已经遍历过的子范围. 避免这一问题就要求 successor 的规范性.

6.6 前向迭代器

如果 successor 是规范的, 算法里就可以多次穿过同一个范围, 也可以在同一范围里维持多个迭代器:

$$\begin{aligned} &\mathit{ForwardIterator}(\mathsf{T}) \triangleq \\ &\qquad \mathit{Iterator}(\mathsf{T}) \\ &\quad \wedge\ \mathsf{regular\_unary\_function}(\mathsf{successor}) \end{aligned}$$

注意, *Iterator* 和 *ForwardIterator* 之间就差一条公理, 而且这里没增加新运算. 除了 successor, 本章后面要介绍的定义在前向迭代器的精化上的其他功能

性过程也都是规范的. successor 的规范性使得在 find_adjacent_mismatch 的实现里向前推进时可以不必保存原值:

```
template<typename I, typename R>
    requires(Readable(I) && ForwardIterator(I) &&
        Relation(R) && ValueType(I) == Domain(R))
I find_adjacent_mismatch_forward(I f, I l, R r)
{
    // 前条件: readable_bounded_range(f, l)
    if (f == l) return l;
    I t;
    do {
        t = f;
        f = successor(f);
    } while (f != l && r(source(t), source(f)));
    return f;
}
```

注意, t 指向不匹配的元素对的第一个元素, 可以返回它.

在第 10 章里, 我们将展示如何利用概念分发 (concept dispatch) 为另一不同的迭代器概念写一个重载的算法版本. 如 _forward 的后缀可以帮助消除不同版本之间的歧义.

successor 的规范性也使得可以为检索划分点实现一个二分段算法:

```
template<typename I, typename P>
    requires(Readable(I) && ForwardIterator(I) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
I partition_point_n(I f, DistanceType(I) n, P p)
{
    // 前条件: readable_counted_range(f, n) ∧ partitioned_n(f, n, p)
    while (!zero(n)) {
        DistanceType(I) h = half_nonnegative(n);
        I m = f + h;
```

```
        if (p(source(m))) {
            n = h;
        } else {
            n = n - successor(h); f = successor(m);
        }
    }
    return f;
}
```

引理 6.9 partition_point_n 返回范围 $[f, n)$ 的 p-划分点.

要在一个有界范围里用二分段法找划分点,[8] 先要得到范围的规模:

```
template<typename I,  typename P>
    requires(Readable(I) && ForwardIterator(I) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
I partition_point(I f, I l, P p)
{
    // 前条件: readable_bounded_range(f, l) ∧ partitioned(f, l, p)
    return partition_point_n(f, l - f, p);
}
```

根据划分点的定义, 立刻可以得到在弱序 r 的一个 r-递增范围里的二分检索算法. 任意值 a, 无论其是否出现在该递增范围里, 都在这种范围里确定了两个迭代器, 分别称为下界 (lower bound) 和上界 (upper bound). 非形式地说, 下界是使等价于 a 的值可能出现在递增序列里的第一个位置. 与之对应, 上界在递增范围里可能出现等价于 a 值的最后一个位置的后继 (successor). 这样, 等价于 a 的元素只能出现在从下界到上界的一个半开范围里. 看一个例子, 假定

8. 二分段技术 (bisection technique) 至少可以追溯到中值定理的证明 Bolzano [1817], 以及与之独立的 Cauchy [1821]. 在 Bolzano 和 Cauchy 用这种技术处理连续函数的最一般情况之前, Lagrange [1795] 早已用它去解决多项式求近似根的特殊问题了. 在检索方面有关二分段法的描述最先由 John W. Mauchly 在其讲稿 "排序和核对" 中给出 [Mauchly 1946].

现在是全序, 相对于值 a 的具有下界 l 和上界 u 的元素具有下面的情况:

$$\underbrace{x_0, x_1, \ldots, x_{l-1}}_{x_i < a}, \underbrace{x_l, \ldots, x_{u-1}}_{x_i = a}, \underbrace{x_u, x_{u+1}, \ldots, x_{n-1}}_{x_i > a}$$

注意, 这三个范围都可以是空的.

引理 6.10 在递增范围 $[f, l)$ 里, 对该范围的值类型里的任意值 a, 下面两个谓词都划分这一范围:

$$\mathsf{lower\_bound}_a(x) \Leftrightarrow \neg r(x, a)$$

$$\mathsf{upper\_bound}_a(x) \Leftrightarrow r(a, x)$$

这就使我们可以将下界和上界形式化地定义为相应谓词的划分点.

引理 6.11 下界迭代器先于或等于上界迭代器.

如果有一个对应于所考虑的谓词的函数对象, 立刻就可以得到下面的确定下界的算法:

```
template<typename R>
    requires(Relation(R))
struct lower_bound_predicate
{
    typedef Domain(R) T;
    const T& a;
    R r;
    lower_bound_predicate(const T& a, R r) : a(a), r(r) { }
    bool operator()(const T& x) { return !r(x, a); }
};

template<typename I, typename R>
    requires(Readable(I) && ForwardIterator(I) &&
        Relation(R) && ValueType(I) == Domain(R))
I lower_bound_n(I f, DistanceType(I) n,
                const ValueType(I)& a, R r)
```

```
{
    // 前条件: weak_ordering(r) ∧ increasing_counted_range(f, n, r)
    lower_bound_predicate<R> p(a, r);
    return partition_point_n(f, n, p);
}
```

上界的处理与此类似:

```
template<typename R>
    requires(Relation(R))
struct upper_bound_predicate
{
    typedef Domain(R) T;
    const T& a;
    R r;
    upper_bound_predicate(const T& a, R r) : a(a), r(r) { }
    bool operator()(const T& x) { return r(a, x); }
};

template<typename I, typename R>
    requires(Readable(I) && ForwardIterator(I) &&
        Relation(R) && ValueType(I) == Domain(R))
I upper_bound_n(I f, DistanceType(I) n,
                const ValueType(I)& a, R r)
{
    // 前条件: weak_ordering(r) ∧ increasing_counted_range(f, n, r)
    upper_bound_predicate<R> p(a, r);
    return partition_point_n(f, n, p);
}
```

练习 6.7 请实现一个过程同时返回下界和上界, 而且其中比较的次数应少于分别独立调用 lower_bound_n 和 upper_bound_n 时的比较次数之和.[9]

9. 一个具有类似功能的 STL 函数称为 equal_range.

在划分点算法里把谓词应用到范围的中点, 可以保证谓词应用次数的最坏情况达到最优. 对任何其他选择, 都可以设想一种情况打败它. 为此只需保证更大的子范围包含划分点就可以了. 如果存在对划分点预期位置的先验知识, 当然可以直接去检查那个点.

partition_point_n 里使用谓词 $\lfloor \log_2 n \rfloor + 1$ 次, 因为每步都使范围的长度除了 2. 这一算法只需要执行对数次的迭代器和整数之间的加法.

引理 6.12 对于前向迭代器, 这一算法执行 successor 运算的总次数小于等于范围的规模.

partition_point 还要计算 $l - f$. 对于前向迭代器, 它还需要加上对 successor 的 n 次调用. 如果应用谓词的代价比调用 successor 的代价更高, 那就值得对前向迭代器做这一操作, 例如对链接表.

引理 6.13 假设划分点的期望距离等于范围的规模的一半, 要用前向迭代器找到划分点, partition_point 将比 find_if 更快一些

$$\mathrm{cost}_{\mathrm{successor}} < \left(1 - 2\frac{\log_2 n}{n}\right) \mathrm{cost}_{\mathrm{predicate}}$$

6.7 索引迭代器

为使 partition_point, lower_bound 和 upper_bound 能主导线性检索的代价, 就必须保证给迭代器加一个整数, 以及做两个迭代器的减法操作都非常快:

IndexedIterator(T) $\triangleq$

ForwardIterator(T)

$\wedge$ $+ : \mathsf{T} \times \mathsf{DistanceType}(\mathsf{T}) \to \mathsf{T}$

$\wedge$ $- : \mathsf{T} \times \mathsf{T} \to \mathsf{DistanceType}(\mathsf{T})$

$\wedge$ $+$ 用常量时间

$\wedge$ $-$ 用常量时间

在 *Iterator* 上定义运算 $+$ 和 $-$ 的基础是 successor. 现在应该把它们作为原语, 而且要高效: 这一概念与 *ForwardIterator* 的差异只是更强的复杂性要求: 希望索引迭代器上的 $+$ 和 $-$ 的代价本质上与 successor 一样.

6.8 双向迭代器

也有些情况中虽然不能直接索引, 但却有反向移动的能力:

$BidirectionalIterator(\mathsf{T}) \triangleq$

$ForwardIterator(\mathsf{T})$

$\wedge$ predecessor : $\mathsf{T} \to \mathsf{T}$

$\wedge$ predecessor 用常量时间

$\wedge$ $(\forall \mathsf{i} \in \mathsf{T})$ successor(i) 有定义 $\Rightarrow$

predecessor(successor(i)) 有定义且等于 i

$\wedge$ $(\forall \mathsf{i} \in \mathsf{T})$ predecessor(i) 有定义 $\Rightarrow$

successor(predecessor(i)) 有定义且等于 i

与 successor 一样, predecessor 也不必是全的; 这个概念的公理将其定义与 successor 的定义相关联. 我们期望 predecessor 的代价在本质上等于 successor 的代价.

引理 6.14 如果 successor 在双向迭代器 i 和 j 上有定义, 那么

$$\mathsf{successor}(\mathsf{i}) = \mathsf{successor}(\mathsf{j}) \Rightarrow \mathsf{i} = \mathsf{j}$$

在一个双向迭代器的弱范围里, 可以反向移动迭代器直到范围的开始:

```
template<typename I>
    requires(BidirectionalIterator(I))
I operator-(I l, DistanceType(I) n)
{
    // 前条件: n ≥ 0 ∧ (∃f ∈ I) weak_range(f, n) ∧ l = f + n
    while (!zero(n)) {
        n = predecessor(n);
        l = predecessor(l);
    }
    return l;
}
```

对双向迭代器可以做反向检索. 正如在前面已经注意到的, 检索一个包含 n 个迭代器的范围时有 n + 1 个可能结果; 前向或反向检索都是如此. 因此我们

约定用 (f,l] 表示左边半开的范围, 用 f 表示 "没找到". 这样做时, 就必须在迭代器 i 处找到满足要求的值时返回 successor(i):

```
template<typename I, typename P>
    requires(Readable(I) && BidirectionalIterator(I) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
I find_backward_if(I f, I l, P p)
{
    // 前条件: (f,l] 是一个可读的有界左半开范围
    while (l != f && !p(source(predecessor(l))))
        l = predecessor(l);
    return l;
}
```

将这一算法与 find_if 比较, 说明了一种程序变换: f 和 l 扮演着互换的角色, source(i) 变成了 source(predecessor(i)), 而且 successor(i) 变成 predecessor(i). 在这一变换下, 在非空范围里 l 可以间接访问, 但 f 却不行.

上面展示的程序变换可用于任何使用前向迭代器的范围的算法. 这也说明有可能实现一个适配器类型, 给它一个双向迭代器类型, 它生成另一个双向迭代器类型, 使其中的 successor 变成 predecessor, predecessor 变成 successor, 且 source 变成 predecessor 的 source.[10] 利用这一适配器类型, 任何基于迭代器或前向迭代器的算法都可以在双向迭代器上反向工作, 它还能让定义在双向迭代器上的任何算法调换遍历的方向.

练习 6.8 重写 find_backward_if, 在循环里只调用 predecessor 一次.

练习 6.9 作为同时用 successor 和 predecessor 的算法实例, 请实现一个谓词确定一个范围是否为回文, 也就是从前向后读和从后向前读都一样的一段文字.

6.9 随机访问迭代器

有些迭代器类型同时满足索引迭代器和双向迭代器的需求. 这类迭代器称为**随机访问迭代器** (random access iterator), 提供计算机寻址的全部功能:

10. 在 STL 里称它为翻转迭代器适配器.

$$
\begin{aligned}
&\mathit{RandomAccessIterator}(\mathsf{T}) \triangleq \\
&\quad \mathit{IndexedIterator}(\mathsf{T}) \wedge \mathit{BidirectionalIterator}(\mathsf{T}) \\
&\wedge\ \mathit{TotallyOrdered}(\mathsf{T}) \\
&\wedge\ (\forall \mathsf{i}, \mathsf{j} \in \mathsf{T})\, \mathsf{i} < \mathsf{j} \Leftrightarrow \mathsf{i} \prec \mathsf{j} \\
&\wedge\ \mathsf{DifferenceType} : \mathit{RandomAccessIterator} \to \mathit{Integer} \\
&\wedge\ + : \mathsf{T} \times \mathsf{DifferenceType}(\mathsf{T}) \to \mathsf{T} \\
&\wedge\ - : \mathsf{T} \times \mathsf{DifferenceType}(\mathsf{T}) \to \mathsf{T} \\
&\wedge\ - : \mathsf{T} \times \mathsf{T} \to \mathsf{DifferenceType}(\mathsf{T}) \\
&\wedge\ < \text{用常量时间} \\
&\wedge\ \text{在迭代器和整数之间的} - \text{运算用常量时间}
\end{aligned}
$$

类型 DifferenceType(T) 应足够大, 足以包含所有距离和它们的加法逆. 如果 i 和 j 是来自一个合法范围的迭代器, $\mathsf{i} - \mathsf{j}$ 总有定义. 同样可以给迭代器加上一个负数, 或者减去一个负数.

对于弱一点的迭代器类型, + 和 − 运算只在一个范围里有定义. 对随机访问迭代器, 除了对 + 和 − 外, 这一事实对 < 也成立. 一般而言, 两个迭代器之间的运算只在它们同属于一个范围时有定义.

项目 6.1 请为随机访问迭代器之间的运算定义与之有关的公理.

由于有下面事实, 我们不需要花很多时间去讨论随机选择迭代器.

定理 6.1 对任何定义在显式地给出的随机访问迭代器范围上的过程, 都存在另一个定义在索引迭代器上的过程与之复杂性相同.

证明. 由于随机访问迭代器上的运算仅在迭代器属于同一范围时有定义, 因此我们可能实现一个适配器类型, 给它一个索引迭代器类型, 它生成一个随机访问迭代器类型. 这种迭代器的状态包含一个迭代器 f 和一个整数 i, 表示迭代器 $\mathsf{f} + \mathsf{i}$. 各种迭代器运算, 例如 +、− 和 <, 都在 i 上运算; source 在 $\mathsf{f} + \mathsf{i}$ 上运算. 换句话说, 就是一个迭代器指向范围的开始, 还有一个到范围内部的索引, 其行为等同于随机访问迭代器. □

这一定理说明, 上述两个概念在已知范围的起点的情况下, 在任意的上下文中都是理论上等价的. 在实践中, 我们也发现采用稍弱的那个概念也不会使性能恶化. 然而, 在一些情况中, 相关的签名需要调整, 需要加进范围的起始点.

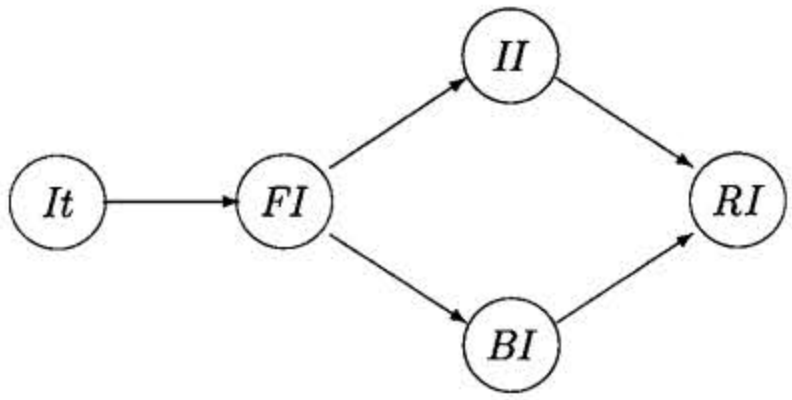


图 6.1 迭代器概念的结构

项目 6.2 请为在一个范围里查找子序列实现一组抽象过程. 说明选择适当算法涉及的各种利弊权衡.[11]

6.10 总结

代数提供了许多概念, 它们具有很好的分层结构, 如半群、模和群. 这些概念使我们可以在更广义的上下文中陈述各种算法. 与此类似, 迭代器概念 (图 6.1) 也使我们可以在最广义的上下文里陈述顺序数据结构上的算法. 这些概念的开发用到了三种精化: 增加一个运算, 或者增强语义, 或者是提出一个更强的复杂性需求. 特别是这里的三个重要概念: 迭代器、前向迭代器和索引迭代器. 它们之间的差异不仅在于与之关联的各种运算, 还在于它们的语义和复杂性. 针对不同迭代器概念的各种各样的检索算法, 计数的和有界的范围, 以及范围的序, 共同构成了顺序程序设计的基础.

11. 本问题的两个广为人知的算法是 Boyer and Moore [1977] 和 Knuth et al. [1977]. Musser and Nishanov [1997] 可以作为这些算法的抽象描述的很好的基础.

第 7 章

坐标结构

第 6 章介绍了一组迭代器概念, 它们可以用作算法与不变的线性形状的对象之间的接口. 本章将超越迭代器继续前进, 讨论处理更复杂的形状的一些坐标结构. 这里将介绍二叉坐标, 并借助一个完成迭代式树遍历的机器实现一些二叉树上的算法. 在讨论了坐标结构的概念模式之后, 最后提出一些与同构、等价和有序相关的算法.

7.1 二叉坐标

迭代器使程序可以遍历线性结构, 其中每个位置只有一个后继. 也存在一些可能有任意多个后继的数据结构. 本章将研究一类重要结构, 其中每个位置恰好有两个后继, 而且分别标明是左后继或者右后继. 为了定义这种结构上的算法, 首先定义下面的概念:

$$
\begin{aligned}
&\mathit{BifurcateCoordinate}(\mathsf{T}) \triangleq \\
&\quad \mathit{Regular}(\mathsf{T}) \\
&\quad \wedge\ \mathsf{WeightType} : \mathit{BifurcateCoordinate} \dashrightarrow \mathit{Integer} \\
&\quad \wedge\ \mathsf{empty} : \mathsf{T} \rightarrow \mathsf{bool} \\
&\quad \wedge\ \mathsf{has\_left\_successor} : \mathsf{T} \rightarrow \mathsf{bool} \\
&\quad \wedge\ \mathsf{has\_right\_successor} : \mathsf{T} \rightarrow \mathsf{bool} \\
&\quad \wedge\ \mathsf{left\_successor} : \mathsf{T} \rightarrow \mathsf{T} \\
&\quad \wedge\ \mathsf{right\_successor} : \mathsf{T} \rightarrow \mathsf{T} \\
&\quad \wedge\ (\forall \mathsf{i}, \mathsf{j} \in \mathsf{T})\,(\mathsf{left\_successor}(\mathsf{i}) = \mathsf{j} \vee \mathsf{right\_successor}(\mathsf{i}) = \mathsf{j}) \Rightarrow \neg\mathsf{empty}(\mathsf{j})
\end{aligned}
$$

这里的 WeightType 类型函数返回一个类型, 它足以用于在通过二叉坐标 (bifurcate coordinate) 遍历的过程中为历经的所有对象计数. 类型 WeightType 类似于迭代器类型的 DistanceType.

这里的谓词 empty 处处有定义. 如果它返回真, 那么其他过程都无定义. empty 是由 has_left_successor 和 has_right_successor 决定的定义空间谓词的否定. has_left_successor 是 left_successor 的定义空间谓词, 而 has_right_successor 是 right_successor 的定义空间谓词. 换一个说法, 如果一个二叉坐标不空, 那么 has_left_successor 和 has_right_successor 就都有定义; 如果这两者中的某个返回真, 与之对应的后继函数就有定义. 针对迭代器的算法需要有一个限界值或计数值标明范围的结束. 而对于二叉结构, 由于存在着很多分支结束的位置, 因此引入谓词 has_left_successor 和 has_right_successor 来确定坐标结构是否有后继就非常自然了.

下面要描述 *BifurcateCoordinate* 概念上的一些算法, 其中的所有运算都是规范的, 这与 *Iterator* 概念的情况有所不同. 出现这种情况, 是因为使用 *Iterator* 的一些最基本算法都不要求 successor 的规范性, 例如 find. 而且那里也确实有一些非规范模型, 例如输入流. 对那些应用了 left_successor 和 right_successor 可能改变基础二叉树形状的结构, 我们需要一个 *WeakBifurcateCoordinate* (弱二叉坐标) 概念, 其中运算不是规范的.

对前面讨论的弱范围, 通过迭代器访问的结构可能是循环的; 而对于计数范围或有界范围, 其形状一定是个线性片段. 与之对应, 为了讨论通过二叉坐标结构访问的结构的形状, 需要有可达性 (reachability) 的概念.

二叉坐标 y 是另一坐标结构 x 的**真后代** (proper descendant), 如果它是 x 的左后继或右后继, 或者它是 x 的左后继或右后继的后代. 二叉坐标结构 y 是坐标结构 x 的**后代** (descendant), 如果 $y = x$, 或者 y 是 x 的真后代.

如果 x 的任何后代 y 都不是它自身的后代, 那么 x 的所有后代就形成一个**有向无环图** (directed acyclic graph, DAG). 换句话说, 要求任意坐标的后继链不会回到它自身. 在这种情况下, x 称为它的后继 DAG 的**根** (root). 如果 x 的后继形成一个 DAG 而且其后继的数目有穷, 它们就构成了一个**有穷** DAG. 有穷 DAG 的**高度** (height)是从它的根出发的最长后继序列的长度加一. 如果 DAG 为空, 其高度就是 0.

如果二叉坐标结构 y 是 x 的左后继的后代, 就说它从 x *左可达* (left reachable), *右可达* (right reachable) 可以类似地定义.

如果 x 的后代形成一个有穷 DAG, 而且对 x 的后代中任意的 y 和 z, 从 y 都不能同时左可达并且右可达 z, 那么 x 的后代就形成了一棵树 (tree). 换句话说, 这要求从一个坐标到它的任意后代只有唯一的一条后继序列. 树的这一性质在本章的算法中所起的作用, 就像有界范围或计数范围的各种性质在第 6 章的作用. 这个性质保证了有穷终止性:

property(C : *BifurcateCoordinate*)

tree : C

x ↦ x 的后代形成一棵树

存在计算树的*权重* (weight, 权) 和高度的递归算法:

```
template<typename C>
    requires(BifurcateCoordinate(C))
WeightType(C) weight_recursive(C c)
{
    // 前条件: tree(c)
    typedef WeightType(C) N;
    if (empty(c)) return N(0);
    N l(0);
    N r(0);
    if (has_left_successor(c))
        l = weight_recursive(left_successor(c));
    if (has_right_successor(c))
        r = weight_recursive(right_successor(c));
    return successor(l + r);
}

template<typename C>
    requires(BifurcateCoordinate(C))
WeightType(C) height_recursive(C c)
{
```

```
    // 前条件: tree(c)
    typedef WeightType(C) N;
    if (empty(c)) return N(0);
    N l(0);
    N r(0);
    if (has_left_successor(c))
        l = height_recursive(left_successor(c));
    if (has_right_successor(c))
        r = height_recursive(right_successor(c));
    return successor(max(l, r));
}
```

引理 7.1 height_recursive(x) ⩽ weight_recursive(x)

height_recursive 也能正确计算 DAG 的高度, 但可能多次访问一些坐标, 访问一个坐标的次数等于到该坐标的路径总数. 这一事实也意味着 weight_recursive 不能正确计算出 DAG 的权重. 遍历 DAG 和有环结构的算法都需要做标记 (marking), 也就是说, 需要采用某种记录各坐标是否已经访问的方法.

深度优先的树遍历存在三种基本顺序. 这三种方法都是在完全遍历了左后代之后再去遍历右后代. *前序* (preorder) 方式在遍历一个坐标的所有后代之前访问这个坐标; *中序* (inorder) 方式在遍历左后代和右后代之间访问该坐标; *后序* (postorder) 方式在遍历了所有后代之后访问该坐标. 我们用下面的类型定义为这三种遍历命名:

```
enum visit { pre, in, post };
```

可以在一个过程里执行上述三种遍历的任意组合, 为此只需让该过程以另一个过程为参数, 由它完成对坐标的访问:

```
template<typename C, typename Proc>
    requires(BifurcateCoordinate(C) &&
        Procedure(Proc) && Arity(Proc) == 2 &&
        visit == InputType(Proc, 0) &&
        C == InputType(Proc, 1))
```

```
Proc traverse_nonempty(C c, Proc proc)
{
    // 前条件: tree(c) ∧ ¬empty(c)
    proc(pre, c);
    if (has_left_successor(c))
        proc = traverse_nonempty(left_successor(c), proc);
    proc(in, c);
    if (has_right_successor(c))
        proc = traverse_nonempty(right_successor(c), proc);
    proc(post, c);
    return proc;
}
```

7.2 双向二叉坐标

递归遍历 (recursive traversal) 需要使用与树的高度成正比的栈空间, 这种空间可能大到等于树中的元素个数; 对于很大而且非平衡的树, 这种空间规模通常无法让人接受. 还有, traverse_nonempty 的接口不允许对多棵树的并发遍历. 一般而言, 在并发地遍历多棵树时, 需要为每棵树使用一个栈. 如果把一个坐标和以前经过的坐标的栈组合起来, 就能得到一种新的坐标类型, 可以通过多做一次变换来获得前驱. (用动作而不是变换可能更高效, 因为可以避免每次拷贝整个栈.) 这种坐标建模了*双向二叉坐标* (bidirectional bifurcate coordinate) 的概念. 这一概念有一个更简单也更灵活的模型: 让树里的每个结点包含一个前驱链接. 这样的树就能支持并发的只需常量空间的遍历, 也能支持各种重新平衡的算法. 因此多用一个额外链接的开销还是合算的.

$$
\begin{aligned}
&\mathit{BidirectionalBifurcateCoordinate}(\mathsf{T}) \triangleq \\
&\qquad \mathit{BifurcateCoordinate}(\mathsf{T}) \\
&\quad \wedge\ \mathsf{has\_predecessor} : \mathsf{T} \to \mathsf{bool} \\
&\quad \wedge\ (\forall \mathsf{i} \in \mathsf{T})\ \neg\mathsf{empty}(\mathsf{i}) \Rightarrow \mathsf{has\_predecessor}(\mathsf{i}) \text{ 有定义} \\
&\quad \wedge\ \mathsf{predecessor} : \mathsf{T} \to \mathsf{T}
\end{aligned}
$$

$\wedge$ ($\forall$i $\in$ T) has_left_successor(i) $\Rightarrow$
predecessor(left_successor(i)) 有定义且等于 i
$\wedge$ ($\forall$i $\in$ T) has_right_successor(i) $\Rightarrow$
predecessor(right_successor(i)) 有定义且等于 i
$\wedge$ ($\forall$i $\in$ T) has_predecessor(i) $\Rightarrow$
is_left_successor(i) $\vee$ is_right_successor(i)

其中的 is_left_successor 和 is_right_successor 定义如下:

```
template<typename T>
    requires(BidirectionalBifurcateCoordinate(T))
bool is_left_successor(T j)
{
    // 前条件: has_predecessor(j)
    T i = predecessor(j);
    return has_left_successor(i) && left_successor(i) == j;
}
```

```
template<typename T>
    requires(BidirectionalBifurcateCoordinate(T))
bool is_right_successor(T j)
{
    // 前条件: has_predecessor(j)
    T i = predecessor(j);
    return has_right_successor(i) && right_successor(i) == j;
}
```

引理 7.2 如果 x 和 y 是双向二叉坐标,

$$\text{left\_successor}(x) = \text{left\_successor}(y) \Rightarrow x = y$$
$$\text{left\_successor}(x) = \text{right\_successor}(y) \Rightarrow x = y$$
$$\text{right\_successor}(x) = \text{right\_successor}(y) \Rightarrow x = y$$

练习 7.1 存在坐标 x 使

$$\text{is\_left\_successor}(x) \wedge \text{is\_right\_successor}(x)$$

这一事实与双向二叉坐标的公理矛盾吗?

无论一个坐标是否有后继, traverse_nonempty 都访问它三次; 维持这一不变式使遍历过程具有一种统一性. 还有, 对一个坐标的三次访问总按同样顺序出现, (pre, in, post), 因此, 给定了当前坐标和刚对它执行的访问, 只要有该坐标及其前驱的信息, 就能确定下一个坐标和下一个状态了. 根据这些认识可以得到一个遍历带有双向二叉坐标的树的迭代式算法, 它只需要用常量的空间. 这种遍历依赖于一部机器 (machine), 也就是说, 一段可以被许多算法作为部件使用的语句序列:

```
template<typename C>
    requires(BidirectionalBifurcateCoordinate(C))
int traverse_step(visit& v, C& c)
{
    // 前条件: has_predecessor(c) ∨ v ≠ post
    switch (v) {
    case pre:
        if (has_left_successor(c))  {
                    c = left_successor(c);  return 1;
        }   v = in;                         return 0;
    case in:
        if (has_right_successor(c)) {
            v = pre; c = right_successor(c); return 1;
        }   v = post;                       return 0;
    case post:
        if (is_left_successor(c))
            v = in;
                    c = predecessor(c);     return -1;
    }
}
```

上面过程的返回值表明了高度的变化. 基于 traverse_step 的算法应该用一个循环, 直至最后一次 (对 post 的) 访问到达最初的坐标时循环终止:

```
template<typename C>
    requires(BidirectionalBifurcateCoordinate(C))
bool reachable(C x, C y)
{
    // 前条件: tree(x)
    if (empty(x)) return false;
    C root = x;
    visit v = pre;
    do {
        if (x == y) return true;
        traverse_step(v, x);
    } while (x != root || v != post);
    return false;
}
```

引理 7.3 如果 reachable 返回, 返回之前 $v = \text{pre}$.

要计算树的权重, 只需统计遍历中 pre 访问的次数:

```
template<typename C>
    requires(BidirectionalBifurcateCoordinate(C))
WeightType(C) weight(C c)
{
    // 前条件: tree(c)
    typedef WeightType(C) N;
    if (empty(c)) return N(0);
    C root = c;
    visit v = pre;
    N n(1); // 不变式: n 是至此访问 pre 的次数
    do {
        traverse_step(v, c);
```

```
        if (v == pre) n = successor(n);
    } while (c != root || v != post);
    return n;
}
```

练习 7.2 请修改 weight, 让它统计 in 或 post 访问的次数, 而不是统计 pre.

要计算树的高度, 算法里需要维护当前高度和经历过的最大值:

```
template<typename C>
    requires(BidirectionalBifurcateCoordinate(C))
WeightType(C) height(C c)
{
    // 前条件: tree(c)
    typedef WeightType(C) N;
    if (empty(c)) return N(0);
    C root = c;
    visit v = pre;
    N n(1); // 不变式: n 是至此 pre 访问中的最大高度
    N m(1); // 不变式: m 是当前 pre 访问的高度
    do {
        m = (m - N(1)) + N(traverse_step(v, c) + 1);
        n = max(n, m);
    } while (c != root || v != post);
    return n;
}
```

额外的 -1 和 $+1$ 是因为 WeightType 是无符号整数. 可以利用 max 的累积版本来帮助改善这段代码.

可以定义一个对应于 traverse_nonempty 的迭代式过程. 这里包含了一个空树检查, 因为不需要在每次递归调用时做这种检查:

```
template<typename C, typename Proc>
    requires(BidirectionalBifurcateCoordinate(C) &&
```

```
        Procedure(Proc) && Arity(Proc) == 2 &&
        visit == InputType(Proc, 0) &&
        C == InputType(Proc, 1))
Proc traverse(C c, Proc proc)
{
    // 前条件: tree(c)
    if (empty(c)) return proc;
    C root = c;
    visit v = pre;
    proc(pre, c);
    do {
        traverse_step(v, c);
        proc(v, c);
    } while (c != root || v != post);
    return proc;
}
```

练习 7.3 利用 traverse_step 和第 2 章的过程, 确定一个双向二叉坐标的后代是否构成一个 DAG.

迭代器的 readable_bounded_range 性质说明了 source 对一个范围里的每个迭代器都有定义. 对二叉坐标的类似性质是

property(C : *Readable*)

requires(BifurcateCoordinate(C))

readable_tree : C

$x \mapsto \mathsf{tree}(x) \wedge (\forall y \in C)\, \mathsf{reachable}(x, y) \Rightarrow \mathsf{source}(y)$ 有定义

要想扩展迭代器算法 (例如 find 和 count) 使之能用于二叉坐标, 存在两种可能的方式: 或是实现一个专用版本, 或是实现一个适配器类型.

项目 7.1 请实现第 6 章的各种算法的双向二叉坐标版本.

项目 7.2 请设计一个适配器类型, 给它一个双向二叉坐标类型, 它能生成一个迭代器类型, 该类型能按照构造迭代器时指定的遍历顺序 (pre、in 或 post) 访

问有关的坐标.

7.3 坐标结构

至此我们已经定义了一些孤立的概念, 针对每个概念给出了一组过程及其语义规程. 定义一种概念模式 (concept schema) 有时也很有意义. 概念模式描述一族概念的某些共同性质. 通常不大可能定义能用于概念模式的算法, 但是对同属一个概念模式的不同概念, 还是有可能描述与之相关的算法的结构. 例如, 前面定义了一些描述线性遍历的迭代器概念, 以及一些描述二叉树遍历的二叉坐标概念. 为能在任意的数据结构里遍历, 现在引进一种称为坐标结构的概念模式. 一个坐标结构可能有若干个相互关联的坐标类型, 每个类型有不同的遍历函数. 坐标结构的概念抽象了数据结构中与漫游有关的方面, 这里考虑的数据结构就是将在第 12 章介绍的组合结构; 它还抽象了存储管理和拥有关系. 完全可以有多种坐标结构描述着同一集对象.

一个特定的概念是一种坐标结构 (coordinate structure), 如果它由一个或几个坐标类型, 0 个或几个值类型, 一个或几个遍历函数, 0 个或几个访问函数组成. 其中每个遍历函数把一个或几个坐标类型和 (或) 值类型映射到一个坐标类型, 每个访问函数把一个或几个坐标类型和 (或) 值类型映射到一个值类型. 例如, 作为一种坐标结构, 一个可读索引迭代器有一个值类型和两个坐标类型, 即该迭代器类型本身及其距离类型. 遍历函数是 + (给一个迭代器加上一个距离) 和 − (给出两个迭代器的距离). 还有一个访问函数 source.

7.4 同构, 等价和有序

同一坐标结构概念的两组坐标称为是同构的 (isomorphic), 如果它们具有同样的形状. 更严格地说, 两组坐标同构的条件是存在它们之间的一个一一对应, 使来自第一组中的一个遍历函数对坐标的任何合法应用返回的坐标, 对应于同一个遍历函数作用于来自第二组里的那个对应坐标得到的结果.

检查同构的算法只使用遍历函数, 因此同构与被坐标指向的对象的值无关. 但同构要求同样的访问函数在对应的一对坐标上或者都有定义, 或者都无定义. 例如, 两个有界或计数范围同构的条件是它们的规模相同. 前向迭代器的

两个弱范围同构的条件是它们具有同样的轨道结构 (有关轨道结构的定义见第 2 章). 两棵树同构的条件是或者它们都为空; 或者它们都不空时被下面代码判断为同构:

```
template<typename C0, typename C1>
    requires(BifurcateCoordinate(C0) &&
        BifurcateCoordinate(C1))
bool bifurcate_isomorphic_nonempty(C0 c0, C1 c1)
{
    // 前条件: tree(c0) ∧ tree(c1) ∧ ¬empty(c0) ∧ ¬empty(c1)
    if (has_left_successor(c0))
        if (has_left_successor(c1)) {
            if (!bifurcate_isomorphic_nonempty(
                    left_successor(c0), left_successor(c1)))
                return false;
        } else return false;
    else if (has_left_successor(c1)) return false;
    if (has_right_successor(c0))
        if (has_right_successor(c1)) {
            if (!bifurcate_isomorphic_nonempty(
                    right_successor(c0), right_successor(c1)))
                return false;
        } else return false;
    else if (has_right_successor(c1)) return false;
    return true;
}
```

引理 7.4 对双向二叉坐标, 树同构的条件是同时遍历得到相同的访问序列:

```
template<typename C0, typename C1>
    requires(BidirectionalBifurcateCoordinate(C0) &&
        BidirectionalBifurcateCoordinate(C1))
bool bifurcate_isomorphic(C0 c0, C1 c1)
```

```
{
    // 前条件: tree(c0) ∧ tree(c1)
    if (empty(c0)) return empty(c1);
    if (empty(c1)) return false;
    C0 root0 = c0;
    visit v0 = pre;
    visit v1 = pre;
    while (true) {
        traverse_step(v0, c0);
        traverse_step(v1, c1);
        if (v0 != v1) return false;
        if (c0 == root0 && v0 == post) return true;
    }
}
```

第 6 章里有一些有关线性和二分段式检索的算法, 它们分别依赖于相等和全序, 而相等和全序又都与规范性概念有关. 在归纳出了取自一种坐标结构的坐标集合上的相等和有序概念之后, 我们就可以去检索对象的集合, 而不仅仅是检索个别的对象了.

来自同一可读坐标结构概念, 并且具有同样值类型的两集坐标称为是按某给定等价关系 (针对一个值类型有一个等价关系) 等价 (equivalent), 如果它们同构, 而且将同一访问函数应用于这两个集合里的对应坐标, 将返回等价的两个对象. 将等价关系换成值类型的相等, 就得到坐标集合相等的一个自然定义.

两个可读有界范围等价的条件是它们的规模相同, 而且对应的迭代器具有相互等价的值:

```
template<typename I0, typename I1, typename R>
    requires(Readable(I0) && Iterator(I0) &&
        Readable(I1) && Iterator(I1) &&
        ValueType(I0) == ValueType(I1) &&
        Relation(R) && ValueType(I0) == Domain(R))
bool lexicographical_equivalent(I0 f0, I0 l0, I1 f1, I1 l1, R r)
```

```
{
    // 前条件: readable_bounded_range(f0, l0)
    // 前条件: readable_bounded_range(f1, l1)
    // 前条件: equivalence(r)
    pair<I0, I1> p = find_mismatch(f0, l0, f1, l1, r);
    return p.m0 == l0 && p.m1 == l1;
}
```

很容易实现一个 lexicographical_equal, 为此只需把一个实现了值相等关系的函数对象传给 lexicographical_equivalent:

```
template<typename T>
    requires(Regular(T))
struct equal
{
    bool operator()(const T& x, const T& y)
    {
        return x == y;
    }
};

template<typename I0, typename I1>
    requires(Readable(I0) && Iterator(I0) &&
        Readable(I1) && Iterator(I1) &&
        ValueType(I0) == ValueType(I1))
bool lexicographical_equal(I0 f0, I0 l0, I1 f1, I1 l1)
{
    return lexicographical_equivalent(f0, l0, f1, l1,
                                      equal<ValueType(I0)>());
}
```

两棵可读树等价的条件是它们同构, 并且相互对应的坐标具有相同的值:

```
template<typename C0, typename C1, typename R>
```

```
    requires(Readable(C0) && BifurcateCoordinate(C0) &&
        Readable(C1) && BifurcateCoordinate(C1) &&
        ValueType(C0) == ValueType(C1) &&
        Relation(R) && ValueType(C0) == Domain(R))
bool bifurcate_equivalent_nonempty(C0 c0, C1 c1, R r)
{
    // 前条件: readable_tree(c0) ∧ readable_tree(c1)
    // 前条件: ¬empty(c0) ∧ ¬empty(c1)
    // 前条件: equivalence(r)
    if (!r(source(c0), source(c1))) return false;
    if (has_left_successor(c0))
        if (has_left_successor(c1)) {
            if (!bifurcate_equivalent_nonempty(
                  left_successor(c0), left_successor(c1), r))
                return false;
        } else return false;
    else if (has_left_successor(c1)) return false;
    if (has_right_successor(c0))
        if (has_right_successor(c1)) {
            if (!bifurcate_equivalent_nonempty(
                  right_successor(c0), right_successor(c1), r))
                return false;
        } else return false;
    else if (has_right_successor(c1)) return false;
    return true;
}
```

对于双向二叉坐标, 两棵树等价的条件是同时访问得到的是同样的访问序列, 而且对应的坐标具有等价的值:

```
template<typename C0, typename C1, typename R>
    requires(Readable(C0) &&
```

```
        BidirectionalBifurcateCoordinate(C0) &&
        Readable(C1) &&
        BidirectionalBifurcateCoordinate(C1) &&
        ValueType(C0) == ValueType(C1) &&
        Relation(R) && ValueType(C0) == Domain(R))
bool bifurcate_equivalent(C0 c0, C1 c1, R r)
{
    // 前条件: readable_tree(c0) ∧ readable_tree(c1)
    // 前条件: equivalence(r)
    if (empty(c0)) return empty(c1);
    if (empty(c1)) return false;
    C0 root0 = c0;
    visit v0 = pre;
    visit v1 = pre;
    while (true) {
        if (v0 == pre && !r(source(c0), source(c1)))
            return false;
        traverse_step(v0, c0);
        traverse_step(v1, c1);
        if (v0 != v1) return false;
        if (c0 == root0 && v0 == post) return true;
    }
}
```

可以用字典序的方式将弱序 (或全序) 扩展到迭代器的可读范围上. 这样做时可忽略等价 (或相等) 的前缀, 并认为更短的范围先于更长的范围:

```
template<typename I0, typename I1, typename R>
    requires(Readable(I0) && Iterator(I0) &&
        Readable(I1) && Iterator(I1) &&
        ValueType(I0) == ValueType(I1) &&
        Relation(R) && ValueType(I0) == Domain(R))
```

```
bool lexicographical_compare(I0 f0, I0 l0, I1 f1, I1 l1, R r)
{
    // 前条件: readable_bounded_range(f0, l0)
    // 前条件: readable_bounded_range(f1, l1)
    // 前条件: weak_ordering(r)
    while (true) {
        if (f1 == l1) return false;
        if (f0 == l0) return true;
        if (r(source(f0), source(f1))) return true;
        if (r(source(f1), source(f0))) return false;
        f0 = successor(f0);
        f1 = successor(f1);
    }
}
```

上述函数可以直截了当地专门化为 lexicographical_less, 为此只需要为 r 传一个表达了值类型上的 < 关系的函数对象:

```
template<typename T>
    requires(TotallyOrdered(T))
struct less
{
    bool operator()(const T& x, const T& y)
    {
        return x < y;
    }
};

template<typename I0, typename I1>
    requires(Readable(I0) && Iterator(I0) &&
        Readable(I1) && Iterator(I1) &&
        ValueType(I0) == ValueType(I1))
bool lexicographical_less(I0 f0, I0 l0, I1 f1, I1 l1)
```

```
{
    return lexicographical_compare(f0, l0, f1, l1,
                                   less<ValueType(I0)>());
}
```

练习 7.4 请解释, 为什么在 lexicographical_compare 里第三个和第四个 if 语句可以交换位置, 但第一个和第二个不能.

练习 7.5 请解释为什么我们不用 find_mismatch 实现 lexicographical_compare.

还可以将字典序扩展到二叉坐标结构, 此时应该忽略等价的有根子树, 并认为一个没有左后继的坐标先于有左后继的坐标. 如果根据当前值和左子树不能确定结果, 就认为没有右后继的坐标先于有右后继的坐标.

练习 7.6 请为可读二叉坐标实现 bifurcate_compare_nonempty.

完成了上面练习的读者, 一定会赞赏下面这个算法的简洁性, 它基于双向坐标, 采用迭代式遍历的方式比较两棵树:

```
template<typename C0, typename C1, typename R>
    requires(Readable(C0) &&
        BidirectionalBifurcateCoordinate(C0) &&
        Readable(C1) &&
        BidirectionalBifurcateCoordinate(C1) &&
        Relation(R) && ValueType(C0) == Domain(R))
bool bifurcate_compare(C0 c0, C1 c1, R r)
{
    // 前条件: readable_tree(c0) ∧ readable_tree(c1) ∧ weak_ordering(r)
    if (empty(c1)) return false;
    if (empty(c0)) return true;
    C0 root0 = c0;
    visit v0 = pre;
    visit v1 = pre;
    while (true) {
        if (v0 == pre) {
```

```
            if (r(source(c0), source(c1))) return true;
            if (r(source(c1), source(c0))) return false;
        }
        traverse_step(v0, c0);
        traverse_step(v1, c1);
        if (v0 != v1) return v0 > v1;
        if (c0 == root0 && v0 == post) return false;
    }
}
```

还可以送给 bifurcate_compare 一个永远失败的关系, 以这种方式来实现 bifurcate_shape_compare. 这将使我们可以对一批树进行排序, 因而可以用 upper_bound 在对数时间里找到其中同构的树.

项目 7.3 请为一族数据结构设计一种坐标结构, 并将同构、等价和序的概念扩展到这种坐标结构.

7.5 总结

线性结构在计算机科学里扮演着基础的角色, 迭代器为这种结构和在其上工作的算法提供了自然的接口. 然而, 也存在一些非线性结构, 它们有自己的非线性的坐标结构. 双向二叉坐标让我们看到了一种迭代式算法, 有关情况与在迭代器范围上的算法大不相同. 我们还把同构、相等和序的概念扩展到了具有不同拓扑结构的坐标集合上.

第 8 章

后继可变的坐标

本章介绍一些迭代器和坐标结构概念, 它们允许重链接, 即针对特定的坐标修改 successor 或遍历函数. 重链接使我们可以实现各种重新安排 (例如排序), 并保持各坐标原来的 source 值. 这里要介绍一些重链接机器, 它们能保持坐标的特定结构性质. 本章最后将总结出一种机器, 它使我们能完成某种特定的树遍历, 其中既不需要栈, 也不需要前驱链接, 而是在遍历中临时性地重链接一些坐标.

8.1 链接迭代器

在第 6 章里, 我们把具体迭代器的 successor 都看成不变的, 因此, 将 successor 应用于特定的迭代器总返回同一个值. **链接迭代器** (linked iterator) 类型是一种前向迭代器类型, 对它存在着一种**链接对象** (linker object). 将这种链接对象作用于一个迭代器可以改变该迭代器的 successor 关系. 链接表建模这种迭代器, 链接表里结点之间的关系可以改变. 这里将使用链接对象, 而不是在迭代器上重载的 set_successor 过程, 这样就能允许同一数据结构的不同链接. 举例说, 双向链表可以通过同时设置后继和前驱链接, 也可以只设置后继链接. 这就使一个多遍算法可以在做最后一遍工作之前不维护前驱关系, 从而尽可能地减少工作量. 通过这种方式, 我们实际上是借助于相应的链接对象, 间接地描述了链接迭代器的概念. 在下面非形式的讨论时, 还是会说链接迭代器类型. 为定义链接对象的需求, 需要定义下面几个与之相关的概念:

$ForwardLinker(S) \triangleq$

IteratorType : $ForwardLinker \to ForwardIterator$

$\wedge$ 令 $\mathrm{I} = \mathsf{IteratorType}(\mathrm{S})$:

$(\forall \mathrm{s} \in \mathrm{S})\,(\mathrm{s} : \mathrm{I} \times \mathrm{I} \to \mathsf{void})$

$\wedge$ $(\forall \mathrm{s} \in \mathrm{S})\,(\forall \mathrm{i}, \mathrm{j} \in \mathrm{I})$ 如果 $\mathsf{successor}(\mathrm{i})$ 有定义,

则 $\mathrm{s}(\mathrm{i}, \mathrm{j})$ 建立起 $\mathsf{successor}(\mathrm{i}) = \mathrm{j}$

$BackwardLinker(\mathrm{S}) \triangleq$

$\mathsf{IteratorType} : BackwardLinker \to BidirectionalIterator$

$\wedge$ 令 $\mathrm{I} = \mathsf{IteratorType}(\mathrm{S})$:

$(\forall \mathrm{s} \in \mathrm{S})\,(\mathrm{s} : \mathrm{I} \times \mathrm{I} \to \mathsf{void})$

$\wedge$ $(\forall \mathrm{s} \in \mathrm{S})\,(\forall \mathrm{i}, \mathrm{j} \in \mathrm{I})$ 如果 $\mathsf{predecessor}(\mathrm{j})$ 有定义,

则 $\mathrm{s}(\mathrm{i}, \mathrm{j})$ 建立起 $\mathrm{i} = \mathsf{predecessor}(\mathrm{j})$

$BidirectionalLinker(\mathrm{S}) \triangleq ForwardLinker(\mathrm{S}) \wedge BackwardLinker(\mathrm{S})$

如果两个范围不包含公共的迭代器, 就说它们不相交 (disjoint). 对于半开的限界范围, 这一情况对应于:

property$(\mathrm{I} : Iterator)$

$\mathsf{disjoint} : \mathrm{I} \times \mathrm{I} \times \mathrm{I} \times \mathrm{I}$

$(\mathrm{f0}, \mathrm{l0}, \mathrm{f1}, \mathrm{l1}) \mapsto (\forall \mathrm{i} \in \mathrm{I})\,\neg(\mathrm{i} \in [\mathrm{f0}, \mathrm{l0}) \wedge \mathrm{i} \in [\mathrm{f1}, \mathrm{l1}))$

另外两类范围的情况都与此类似. 由于链接迭代器也是迭代器, 因此也遵循我们为范围定义的各种概念. 但是, 对于链接迭代器, 范围的不相交和其他性质都可能随着时间的变化而改变. 完全可能有这种情况, 只有一个前向链接的前向迭代器 (单链表) 的若干互不相交的范围共享同一个极限, 通常称为 *nil*.

8.2 链接重整

一个**链接重整** (link rearrangement) 是一个算法, 它以一个或多个链接范围为参数, 返回一个或多个链接范围, 并满足如下性质.

- 不同输入范围 (无论计数的还是限界的) 两两不相交.
- 不同输出范围 (无论计数的还是限界的) 两两不相交.
- 每一个输入范围里的每个迭代器都出现在某一个输出范围里面.

- 每一个输出范围里的每个迭代器都出现在某一个输入范围里面.
- 任何输出范围里的任一迭代器所指的对象都与重整之前一样, 而且该对象的值也没有改变.

还请注意, 在输入范围里成立的 successor 和 predecessor 关系, 在输出范围里未必还成立.

一个**链接重整保持先于关系** (precedence preserving), 如果在任一输出范围里的来自同一个输入范围的两个迭代器有 $i \prec j$, 那么关系 $i \prec j$ 在原输入范围里也一定成立.

要实现一个链接重整, 需要很小心地满足上面提出的不相交性、保持性和序关系. 下面我们将首先展示三个很短的过程 (或说是机器), 其中每个都执行一步遍历或者链接. 而后从这些机器出发组合出各种链接重整, 完成链接范围的划分、组合和翻转. 前两个机器的功能是建立或维持通过引用传来的两个迭代器之间的关系 f = successor(t):

```
template<typename I>
    requires(ForwardIterator(I))
void advance_tail(I& t, I& f)
{
    // 前条件: successor(f) 有定义
    t = f;
    f = successor(f);
}

template<typename S>
    requires(ForwardLinker(S))
struct linker_to_tail
{
    typedef IteratorType(S) I;
    S set_link;
    linker_to_tail(const S& set_link) : set_link(set_link) { }
    void operator()(I& t, I& f)
```

```
    {
        // 前条件: successor(f) 有定义
        set_link(t, f);
        advance_tail(t, f);
    }
};
```

可以用 advance_tail 找到非空限界范围里的最后一个迭代器:[1]

```
template<typename I>
    requires(ForwardIterator(I))
I find_last(I f, I l)
{
    // 前条件: bounded_range(f,l) ∧ f ≠ l
    I t;
    do
        advance_tail(t, f);
    while (f != l);
    return t;
}
```

可以结合使用 advance_tail 和 linker_to_tail, 根据将一个*伪谓词* (pseudo predicate) 应用到范围里的各个迭代器上的结果, 把该范围分划为两个范围. *伪谓词*不必是规范的, 其结果可以依赖于它的自身状态或者输入. 举例说, 一个伪谓词可以忽略其参数而简单地交替返回假和真. 本算法有三个参数: 一个链接迭代器的限界范围, 一个定义在链接迭代器类型上的伪谓词, 还有一个链接对象. 算法返回一对范围, 它们分别包含了不满足伪谓词的那些迭代器和满足它的那些迭代器. 把这样返回的范围表达为闭限界范围 [h, t] 更加方便, 这里的 h 是第一个 (或称为*头*, head) 迭代器, 而 t 是最后一个 (或称为*尾*, tail) 迭代器. 返回每个范围的尾, 使调用者可以重新链接起迭代器, 而不必再做一次遍历去找到它 (不必用例如 find_last). 当然, 返回的两个范围都可能为空, 这里通过返回 h = t = l 表示, 其中的 l 是输入范围的极限. 本算法不修改返回的两个范围里

1. 应该看到, 第 6 章里的 find_adjacent_mismatch_forward 隐含地使用了 advance_tail.

的尾迭代器的 successor. 下面是这一算法:

```
template<typename I, typename S, typename Pred>
    requires(ForwardLinker(S) && I == IteratorType(S) &&
        UnaryPseudoPredicate(Pred) && I == Domain(Pred))
pair< pair<I, I>, pair<I, I> >
split_linked(I f, I l, Pred p, S set_link)
{
    // 前条件: bounded_range(f,l)
    typedef pair<I, I> P;
    linker_to_tail<S> link_to_tail(set_link);
    I h0 = l; I t0 = l;
    I h1 = l; I t1 = l;
    if (f == l)                                  goto s4;
    if (p(f)) { h1 = f; advance_tail(t1, f); goto s1; }
    else      { h0 = f; advance_tail(t0, f); goto s0; }
s0: if (f == l)                                  goto s4;
    if (p(f)) { h1 = f; advance_tail(t1, f); goto s3; }
    else      {         advance_tail(t0, f); goto s0; }
s1: if (f == l)                                  goto s4;
    if (p(f)) {         advance_tail(t1, f); goto s1; }
    else      { h0 = f; advance_tail(t0, f); goto s2; }
s2: if (f == l)                                  goto s4;
    if (p(f)) {         link_to_tail(t1, f); goto s3; }
    else      {         advance_tail(t0, f); goto s2; }
s3: if (f == l)                                  goto s4;
    if (p(f)) {         advance_tail(t1, f); goto s3; }
    else      {         link_to_tail(t0, f); goto s2; }
s4: return pair<P, P>(P(h0, t0), P(h1, t1));
}
```

这个过程也是一个状态机. 其中的变量 t0 和 t1 分别指向两个输出范围的尾. 机器的状态对应于下面几个条件:

s0: $\mathsf{successor}(\mathrm{t0}) = \mathrm{f} \wedge \neg \mathrm{p}(\mathrm{t0})$

s1: $\mathsf{successor}(\mathrm{t1}) = \mathrm{f} \wedge \mathrm{p}(\mathrm{t1})$

s2: $\mathsf{successor}(\mathrm{t0}) = \mathrm{f} \wedge \neg \mathrm{p}(\mathrm{t0}) \wedge \mathrm{p}(\mathrm{t1})$

s3: $\mathsf{successor}(\mathrm{t1}) = \mathrm{f} \wedge \neg \mathrm{p}(\mathrm{t0}) \wedge \mathrm{p}(\mathrm{t1})$

只有在状态 s2 和 s3 之间转换时才需要重新链接. 在上面算法里, 即使某状态的下一状态紧随其后, 也写了 goto 语句, 这样做只是为了看起来对称.

引理 8.1 对于 split_linked 返回的每个 $[\mathrm{h}, \mathrm{t}]$, 都有 $\mathrm{h} = \mathrm{l} \Leftrightarrow \mathrm{t} = \mathrm{l}$.

练习 8.1 假定 split_linked 返回的一个范围 (h, t) 不空, 请解释 t 指向哪里, $\mathsf{successor}(\mathrm{t})$ 的值是什么.

引理 8.2 split_linked 是一个保持先于关系的链接重整.

通过将一个伪关系作用到两个输入范围的剩余部分的头, 可以利用 advance_tail 和 linker_to_tail 实现一个算法把两个范围合而为一. *伪关系* (pseudo relation) 就是同源的二元伪谓词, 不必是规范的. 这个算法有四个参数: 两个链接迭代器的范围, 一个定义在相应迭代器类型上的伪关系, 以及一个链接对象. 算法返回一个三元组 $(\mathrm{f}, \mathrm{t}, \mathrm{l})$, 其中 $[\mathrm{f}, \mathrm{l})$ 是组合起来的迭代器的一个半开范围, $\mathrm{t} \in [\mathrm{f}, \mathrm{l})$ 是最后访问的迭代器. 随后调用 $\mathsf{find\_last}(\mathrm{t}, \mathrm{l})$ 将返回该范围里的最后一个迭代器, 使得到的范围可以链接另一个范围. 下面是这个算法:

```
template<typename I, typename S, typename R>
    requires(ForwardLinker(S) && I == IteratorType(S) &&
        PseudoRelation(R) && I == Domain(R))
triple<I, I, I>
combine_linked_nonempty(I f0, I l0, I f1, I l1, R r, S set_link)
{
    // 前条件: bounded_range(f0, l0) ∧ bounded_range(f1, l1)
    // 前条件: f0 ≠ l0 ∧ f1 ≠ l1 ∧ disjoint(f0, l0, f1, l1)
```

```
    typedef triple<I, I, I> T;
    linker_to_tail<S> link_to_tail(set_link);
    I h; I t;
    if (r(f1, f0)) { h = f1; advance_tail(t, f1); goto s1; }
    else           { h = f0; advance_tail(t, f0); goto s0; }
s0: if (f0 == l0)                                 goto s2;
    if (r(f1, f0)) {         link_to_tail(t, f1); goto s1; }
    else           {         advance_tail(t, f0); goto s0; }
s1: if (f1 == l1)                                 goto s3;
    if (r(f1, f0)) {         advance_tail(t, f1); goto s1; }
    else           {         link_to_tail(t, f0); goto s0; }
s2: set_link(t, f1); return T(h, t, l1);
s3: set_link(t, f0); return T(h, t, l0);
}
```

练习 8.2 请实现 combine_linked, 它允许空的输入. 这时应该返回什么作为最后访问的迭代器?

上面过程也是一个状态机. 变量 t 指向输出范围的尾. 其中的状态对应于下面两个条件:

s0: $\mathsf{successor}(\mathrm{t}) = \mathrm{f0} \wedge \neg \mathrm{r}(\mathrm{f1}, \mathrm{t})$

s1: $\mathsf{successor}(\mathrm{t}) = \mathrm{f1} \wedge \mathrm{r}(\mathrm{t}, \mathrm{f0})$

只有在 s0 和 s1 之间转换时需要重新链接.

引理 8.3 如果调用 $\mathsf{combine\_linked\_nonempty}(\mathrm{f0}, \mathrm{l0}, \mathrm{f1}, \mathrm{l1}, \mathrm{r}, \mathrm{s})$ 返回 $(\mathrm{h}, \mathrm{t}, \mathrm{l})$, 那么其中的 h 等于 f0 或 f1, 而且与之独立地有 l 等于 l0 或 l1.

引理 8.4 在到达状态 s2 时 t 位于原范围 $[\mathrm{f0}, \mathrm{l0})$ 里, $\mathsf{successor}(\mathrm{t}) = \mathrm{l0}$ 且 $\mathrm{f1} \neq \mathrm{l1}$. 当到达状态 s3 时, t 位于原范围 $[\mathrm{f1}, \mathrm{l1})$ 里, $\mathsf{successor}(\mathrm{t}) = \mathrm{l1}$ 且 $\mathrm{f0} \neq \mathrm{l0}$.

引理 8.5 combine_linked_nonempty 是保持先于关系的链接重整.

第三个机器链接一个表的头, 而不是尾:

```
template<typename I, typename S>
    requires(ForwardLinker(S) && I == IteratorType(S))
struct linker_to_head
{
    S set_link;
    linker_to_head(const S& set_link) : set_link(set_link) { }
    void operator()(I& h, I& f)
    {
        // 前条件: successor(f) 有定义
        IteratorType(S) tmp = successor(f);
        set_link(f, h);
        h = f;
        f = tmp;
    }
};
```

借助于这一机器, 可以实现一个迭代器范围的翻转:

```
template<typename I, typename S>
    requires(ForwardLinker(S) && I == IteratorType(S))
I reverse_append(I f, I l, I h, S set_link)
{
    // 前条件: bounded_range(f, l) ∧ h ∉ [f, l)
    linker_to_head<I, S> link_to_head(set_link);
    while (f != l) link_to_head(h, f);
    return h;
}
```

为了避免共享的真尾部, h 应该是另一个不相交的表的头 (等于单链表, nil 也可以接受), 或者是 l. 由于 l 可能已经被用作 h 的初值 (这样给出的是 reverse_linked), 另外送一个独立的累积参数也很有用.

8.3 链接重整的应用

给定了某链接迭代器类型的值类型上的一个谓词, 我们可以用 split_linked 来划分范围. 为此需要有一个适配器, 把值上的谓词转变为迭代器上的谓词:

```
template<typename I, typename P>
    requires(Readable(I) &&
        Predicate(P) && ValueType(I) == Domain(P))
struct predicate_source
{
    P p;
    predicate_source(const P& p) : p(p) { }
    bool operator()(I i)
    {
        return p(source(i));
    }
};
```

有了这个适配器, 就可以把一个范围划分为一个值不满足给定谓词的范围, 以及另一个值满足给定谓词的范围了:

```
template<typename I, typename S, typename P>
    requires(ForwardLinker(S) && I == IteratorType(S) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
pair< pair<I, I>, pair<I, I> >
partition_linked(I f, I l, P p, S set_link)
{
    predicate_source<I, P> ps(p);
    return split_linked(f, l, ps, set_link);
}
```

给定了某个链接迭代器类型的值类型上的一个弱序, 我们就可以借助 combine_linked_nonempty 来归并两个递增的范围. 同样的, 这里也需要一个适配器, 用于把值上的关系转换为迭代器上的关系:

```
template<typename I0, typename I1, typename R>
    requires(Readable(I0) && Readable(I1) &&
        ValueType(I0) == ValueType(I1) &&
        Relation(R) && ValueType(I0) == Domain(R))
struct relation_source
{
    R r;
    relation_source(const R& r) : r(r) { }
    bool operator()(I0 i0, I1 i1)
    {
        return r(source(i0), source(i1));
    }
};
```

用这一关系组合了两个范围之后,剩下的工作就是找出组合范围的最后一个迭代器,并将它设置到 l1:

```
template<typename I, typename S, typename R>
    requires(Readable(I) &&
        ForwardLinker(S) && I == IteratorType(S) &&
        Relation(R) && ValueType(I) == Domain(R))
pair<I, I> merge_linked_nonempty(I f0, I l0, I f1, I l1,
                                 R r, S set_link)
{
    // 前条件: f0 ≠ l0 ∧ f1 ≠ l1
    // 前条件: increasing_range(f0, l0, r)
    // 前条件: increasing_range(f1, l1, r)
    relation_source<I, I, R> rs(r);
    triple<I, I, I> t = combine_linked_nonempty(f0, l0, f1, l1,
                                                rs, set_link);
    set_link(find_last(t.m1, t.m2), l1);
    return pair<I, I>(t.m0, l1);
```

```
}
```

引理 8.6 若 [f0, l0) 和 [f1, l1) 都是非空的递增限界范围, 用 merge_linked_nonempty 归并它们将得到一个递增的范围.

引理 8.7 如果 i0 ∈ [f0, l0) 和 i1 ∈ [f1, l1) 是两个迭代器, 其值在 r 下等价, 用 merge_linked_nonempty 归并这两个范围后 i0 ≺ i1.

有了 merge_linked_nonempty, 立刻就可以实现归并排序:

```
template<typename I, typename S, typename R>
    requires(Readable(I) &&
        ForwardLinker(S) && I == IteratorType(S) &&
        Relation(R) && ValueType(I) == Domain(R))
pair<I, I> sort_linked_nonempty_n(I f, DistanceType(I) n,
                                  R r, S set_link)
{
    // 前条件: counted_range(f, n) ∧ n > 0 ∧ weak_ordering(r)
    typedef DistanceType(I) N;
    typedef pair<I, I> P;
    if (n == N(1)) return P(f, successor(f));
    N h = half_nonnegative(n);
    P p0 = sort_linked_nonempty_n(f, h, r, set_link);
    P p1 = sort_linked_nonempty_n(p0.m1, n - h, r, set_link);
    return merge_linked_nonempty(p0.m0, p0.m1,
                                 p1.m0, p1.m1, r, set_link);
}
```

引理 8.8 sort_linked_nonempty_n 是一个链接重整.

引理 8.9 如果 ⟦f, n⟩ 是非空的计数范围, sort_linked_nonempty_n 将把它重整为一个递增的限界范围.

一个链接范围的相对于某个弱序 r 的排序是**稳定的** (stability), 如果在输入范围中有两个具有关系 i ≺ j 的迭代器, 而且它们的值按关系 r 等价, 那么在

输出中必有 i ≺ j.

引理 8.10 sort_linked_nonempty_n 相对于所提供的弱序 r 是稳定的.

练习 8.3 请设法确定 sort_linked_nonempty_n 里的关系和链接对象应用的最大和平均次数.

虽然 sort_linked_nonempty_n 中的运算执行次数接近最优, 如果被排序的链接结构太大, 因而不能放入缓存, *引用局部性* (locality of reference) 方面的糟糕情况也会限制这个算法的有用性. 在这种情况下, 如果存在可用的额外存储, 就应该把链接表拷贝到一个数组里, 然后在数组里排序.

链接表排序并不要求 predecessor 的值. 如果把下式当作不变式

$$\mathrm{i} = \mathsf{predecessor}(\mathsf{successor}(\mathrm{i}))$$

为维持它而做的反向链接运算的次数将正比于比较的次数. 我们可以通过临时打破上面不变式的方式避免这种额外工作. 假设 I 是一个链接的双向迭代器类型, 而 forward_linker 和 backward_linker 分别是 I 的前向和反向链接对象. 我们可以把 forward_linker 送给排序过程, 也就是说, 把这个表当作一个单链表处理, 最后再通过对第一个之后的每个迭代器应用 backward_linker 来恢复 predecessor 链接:

```
pair<I, I> p = sort_linked_nonempty_n(f, n,
                                      r, forward_linker);
f = p.m0;
while (f != p.m1) {
    backward_linker(f, successor(f));
    f = successor(f);
};
```

练习 8.4 请实现一个保持先行关系的链接重整过程 unique, 它以一个链接范围和定义在迭代器的值类型上的一个等价关系为参数, 产生出两个范围. 如果紧随一个迭代器之后的一些迭代器具有与之等价的值, 该过程把这样的迭代器序列都放到第二个范围里.

8.4 链接的二叉坐标

允许修改 successor 产生了各种链接重整算法, 例如组合和划分. 让其他坐标结构具有可变的遍历函数也很有用. 现在用链接二叉坐标来阐释这方面的想法.

对链接迭代器, 前面是把链接运算作为参数传给它, 这是因为可能需要不同的链接运算. 例如, 在排序之后可能需要重新恢复反向链接. 对链接的二叉坐标, 至今尚未看到需要用不同版本的链接运算的情况, 所以采用在概念里定义它们的方式:

$LinkedBifurcateCoordinate(\mathsf{T}) \triangleq$

$\quad BifurcateCoordinate(\mathsf{T})$

$\wedge$ set_left_successor : $\mathsf{T} \times \mathsf{T} \to$ void

$\quad (\mathsf{i}, \mathsf{j}) \mapsto$ 建立 left_successor(i) = j

$\wedge$ set_right_successor : $\mathsf{T} \times \mathsf{T} \to$ void

$\quad (\mathsf{i}, \mathsf{j}) \mapsto$ 建立 right_successor(i) = j

set_left_successor 和 set_right_successor 的定义空间是非空坐标的集合.

树的概念衍生出很丰富的一集有用数据结构和算法. 作为本章的一个总结, 下面将给出一小组算法, 其中展示了一种很重要的编程技术. 该技术称为**链接反转** (link reversal), 用于在遍历树的过程中修改链接, 完成遍历之后恢复原来的状态, 其间只需要常量的额外空间. 链接反转需要另外的公理, 以便处理空坐标的情况. 遍历函数对这种坐标无定义:

$EmptyLinkedBifurcateCoordinate(\mathsf{T}) \triangleq$

$\quad LinkedBifurcateCoordinate(\mathsf{T})$

$\wedge$ empty(T())[2]

$\wedge \neg$empty(i) $\Rightarrow$

$\quad$ left_successor(i) 和 right_successor(i) 有定义

$\wedge \neg$empty(i) $\Rightarrow$

$\quad (\neg$has_left_successor(i) $\Leftrightarrow$ empty(left_successor(i)))

$\wedge \neg$empty(i) $\Rightarrow$

$\quad (\neg$has_right_successor(i) $\Leftrightarrow$ empty(right_successor(i)))

2. 换句话说, empty 对默认构造的值为真, 可能对另外一些值也为真.

第 7 章介绍的 traverse_step 是遍历双向二叉坐标的一种有效方法, 但它需要用到 predecessor 函数. 如果 predecessor 函数不可用, 而且由于树的不平衡, (基于栈的) 递归遍历又无法接受, 那么就可以采用下面的链接反转技术, 在一个平常用于保存后继的链接中临时性地存一下到前驱的链接, 以保证存在一条回到根的路径.[3]

如果考虑一个树结点的左后继和右后继, 再加上前一树结点的坐标构成的三元组, 用下面机器可以完成这个三元组中三个成员的轮换:

```
template<typename C>
    requires(EmptyLinkedBifurcateCoordinate(C))
void tree_rotate(C& curr, C& prev)
{
    // 前条件: ¬empty(curr)
    C tmp = left_successor(curr);
    set_left_successor(curr, right_successor(curr));
    set_right_successor(curr, prev);
    if (empty(tmp)) { prev = tmp; return; }
    prev = curr;
    curr = tmp;
}
```

反复应用 tree_rotate 就能遍历整个树:

```
template<typename C, typename Proc>
    requires(EmptyLinkedBifurcateCoordinate(C) &&
        Procedure(Proc) && Arity(Proc) == 1 &&
        C == InputType(Proc, 0))
Proc traverse_rotating(C c, Proc proc)
{
    // 前条件: tree(c)
```

3. 链接反转技术由 Schorr and Waite [1967] 引进, L. P. Deutsch 独立地发现了这种技术. 没有标志位的技术是 Robson [1973] 和 Morris [1979] 发表的. 我们在这里给出的特殊的链接轮换技术应归功于 Lindstrom [1973], 还有 Dwyer [1974] 的独立工作.

```
    if (empty(c)) return proc;
    C curr = c;
    C prev;
    do {
        proc(curr);
        tree_rotate(curr, prev);
    } while (curr != c);
    do {
        proc(curr);
        tree_rotate(curr, prev);
    } while (curr != c);
    proc(curr);
    tree_rotate(curr, prev);
    return proc;
}
```

定理 8.1 考虑 traverse_rotating(c, proc) 的一次调用和 c 的任何非空后代 i, 其中 i 有初始的左右后继 l 和 r 以及前驱 p. 那么

1. i 的左右后继历经三次迁移:

$$(l, r) \overset{pre}{\to} (r, p) \overset{in}{\to} (p, l) \overset{post}{\to} (l, r)$$

2. 如果 n_l 和 n_r 是 l 和 r 的权重, 迁移 $(r, p) \overset{in}{\to} (p, l)$ 和 $(p, l) \overset{post}{\to} (l, r)$ 分别做 $3n_l + 1$ 和 $3n_r + 1$ 次 tree_rotate 调用.

3. 如果 k 是 tree_rotate 调用的运行计数器, 在 i 的后继的三次迁移时, k mod 3 的值各不相同.

4. 在调用 traverse_rotating(c, proc) 期间, 调用 tree_rotate 的总次数是 3n, 其中 n 是 c 的权重.

证明. 对 n 和 c 的权重做归纳. □

练习 8.5 请画出用 traverse_rotating 遍历一棵 7 个结点的完全二叉树的过程中历经的每个状态.

traverse_rotating 将执行与第 7 章的 traverse_nonempty 一样的前序、中序和后序序列. 但是很可惜, 我们不知道如何确定对某坐标的一次特定访问究竟是 pre, in, 还是 post 访问. 然而, 还是可以利用 traverse_rotating 计算一些有用的东西, 例如一棵树的权重:

```
template<typename T, typename N>
    requires(Integer(N))
struct counter {
    N n;
    counter() : n(0) { }
    counter(N n) : n(n) { }
    void operator()(const T&) { n = successor(n); }
};

template<typename C>
    requires(EmptyLinkedBifurcateCoordinate(C))
WeightType(C) weight_rotating(C c)
{
    // 前条件: tree(c)
    typedef WeightType(C) N;
    return traverse_rotating(c, counter<C, N>()).n / N(3);
}
```

利用访问计数值取模 3, 就可以访问每个坐标恰好一次:

```
template<typename N, typename Proc>
    requires(Integer(N) &&
        Procedure(Proc) && Arity(Proc) == 1)
struct phased_applicator
{
    N period;
    N phase;
    N n;
```

```
    // 不变式: n, phase ∈ [0, period)
    Proc proc;
    phased_applicator(N period, N phase, N n, Proc proc) :
        period(period), phase(phase), n(n), proc(proc) { }
    void operator()(InputType(Proc, 0) x)
    {
        if (n == phase) proc(x);
        n = successor(n);
        if (n == period) n = 0;
    }
};

template<typename C, typename Proc>
    requires(EmptyLinkedBifurcateCoordinate(C) &&
        Procedure(Proc) && Arity(Proc) == 1 &&
        C == InputType(Proc, 0))
Proc traverse_phased_rotating(C c, int phase, Proc proc)
{
    // 前条件: tree(c) ∧ 0 ⩽ phase < 3
    phased_applicator<int, Proc> applicator(3, phase, 0, proc);
    return traverse_rotating(c, applicator).proc;
}
```

项目 8.1 考虑用 tree_rotate 实现二叉树上的同构、等价和有序.

8.5 结论

带有可变遍历函数的链接坐标结构可以支持各种有用的重整算法, 例如链接范围的排序. 从一个简单的像机器一样的组件组合出这类算法, 可以得到具有精确的数学性质的有效代码. 按一定的规矩使用 goto 是实现状态机的一种正统方式. 如果一个不变式涉及多个对象, 在更新这些对象之一的期间有可能需要临时性地违背不变式. 一个算法定义了一个区域, 可以允许在其内部临时地

违背不变式. 只要能保证在退出这种区域之前恢复不变式, 就允许出现暂时违背的情况.

第 9 章

拷贝

本章介绍可写迭代器，其访问函数允许修改迭代器的值. 这里用一族拷贝算法来展示可写迭代器的使用，这些算法的基础是一个简单机器，其功能就是拷贝一个对象并更新其输入和输出迭代器. 仔细描述的前条件将允许输入和输出范围在拷贝期间部分重叠. 如果两个同样大小的不重叠范围都允许修改，那么就可以用一些对换算法去交换它们的内容.

9.1 可写性

本章讨论迭代器和其他坐标结构的第二类访问: 可写性. 一个类型称为可写的，如果一元过程 sink 在其上有定义. sink 只能用在赋值的左部，赋值的右部求值得到的应是类型 ValueType(T) 的一个对象:

$$
\begin{aligned}
&\mathit{Writable}(\mathsf{T}) \triangleq \\
&\quad \mathsf{ValueType} : \mathit{Writable} \to \mathit{Regular} \\
&\quad \wedge\ (\forall \mathsf{x} \in \mathsf{T})\,(\forall \mathsf{v} \in \mathsf{ValueType}(\mathsf{T}))\,\mathsf{sink}(\mathsf{x}) \leftarrow \mathsf{v} \text{ 是良构的语句}
\end{aligned}
$$

sink(x) 的符合 *Writable* 概念的使用只有放在赋值左部. 当然，完全可以有某些特殊类型的 *Writable* 的模型支持其他使用方式.

sink 不必是全的，在一个可写类型里完全可能存在一些 sink 无定义的对象. 就像可读性的情况一样，可写性概念也不提供定义空间谓词来确定 sink 是否对某个特定对象有定义. 在算法里使用 sink 的合法性只能根据前条件确定.

对于对象 x 的一个特定状态，*Writable* 概念只保证对 sink(x) 的一次赋值是合法的. 有可能存在某些特定类型的 *Writable*，它们提供了某种规程，允许对

sink(x) 的后续赋值.[1]

可写对象 x 和可读对象 y 互为*别名* (aliased) 的条件是 sink(x) 和 source(y) 两者都有定义, 而且无论把什么值 v 赋给了 sink(x), 都会导致它作为 source(y) 的值反应出来:

property(T : *Writable*, U : *Readable*)
  requires(ValueType(T) = ValueType(U))
aliased : T × U
  (x, y) ↦ sink(x) 有定义 ∧
    source(y) 有定义 ∧
    (∀v ∈ ValueType(T)) sink(x) ← v 建立 source(y) = v

最后一类访问称为*可变动* (mutable), 它是可读性和可写性的一种合理的组合方式:

Mutable(T) ≜
  Readable(T) ∧ *Writable*(T)
  ∧ (∀x ∈ T) sink(x) 有定义 ⇔ source(x) 有定义
  ∧ (∀x ∈ T) sink(x) 有定义 ⇒ aliased(x, x)
  ∧ deref : T → ValueType(T)&
  ∧ (∀x ∈ T) sink(x) 有定义 ⇔ deref(x) 有定义

对可变动迭代器, 将 source(x) 或 sink(x) 换为 deref(x) 不会影响程序执行的意义.

同时建模 *Writable* 和 *Iterator* 的某迭代器类型的一个范围称为是*可写范围* (writable range), 如果 sink 对该范围里的所有迭代器都有定义:

property(I : *Writable*)
  requires(*Iterator*(I))
writable_bounded_range : I × I
  (f, l) ↦ bounded_range(f, l) ∧ (∀i ∈ [f, l)) sink(i) 有定义

可以类似地定义 writable_weak_range 和 writable_counted_range .

---

1. Jerry Schwarz 提出了一种可能更优美的接口: 把 sink 换成过程 store, 使 store(v, x) 等价于 sink(x) ← v.

允许对可读迭代器 i 多次调用 source(i), 而且保证这些调用总返回同一个值. 也就是说, 这里的 source 是规范的. 这一规范性使我们可能写出一些简单而且有用的算法, 例如 find_if. 然而, 对可写迭代器 j 而言, 对 sink(j) 的赋值就不是可重复的: 通过一个迭代器的两次赋值之间必须有一个 successor 调用分隔. 可读和可写迭代器之间的不对称性符合我们的需要: 看来它不会排除有用的算法, 同时又能允许一些模型, 例如非缓冲的输出流. 在 *Iterator* 概念里非规范的 successor 和非规范的 sink 可以支持一些算法, 它们不仅可以用于内存数据结构, 还可以用于输入流和输出流.

同时建模 *Mutable* 和 *ForwardIterator* 某迭代器类型的一个范围称为可变动范围 (mutable range), 如果 sink、source 和 deref 对该范围里的所有迭代器都有定义. 只有多遍的算法会对同一个范围的迭代器既读又写. 这样, 对可变动范围, 我们至少需要前向迭代器, 两次赋值之间也不再要求有 successor 调用:

property(I : *Mutable*)
  requires(*ForwardIterator*(I))
mutable_bounded_range : I × I
  (f, l) ↦ bounded_range(f, l) ∧ (∀i ∈ [f, l)) sink(i) 有定义

mutable_weak_range 和 mutable_counted_range 可以类似定义.

9.2 基于位置的拷贝

现在要给出一族算法, 它们从一个或几个输入范围拷贝对象到一个或几个输出范围. 一般而言, 这些算法的后条件都要描述输出范围里的对象与输入范围里的对象的原值相等. 当输入范围和输出范围不重叠时很容易建立所期望的后条件. 然而, 在存在重叠的范围之间拷贝对象也很有用, 这里每个算法的后条件都需要描述所允许的重叠情况.

如果输入范围里的一个迭代器与输出范围里的一个迭代器互为别名, 就要遵循有关重叠的基本规则: 算法不能在将 sink 作用于给定的输出迭代器之后, 再将 source 也作用于相应的输入迭代器. 在下面给出的各种算法里, 都需要开发出一些精确的条件和一般的性质来表述这方面的情况.

下面将基于一些机器组合起各种拷贝算法, 这些机器都引用了两个迭代器, 它们不但负责拷贝, 也负责更新这些迭代器. 最常用的一部机器完成一个对象的拷贝, 还将两个迭代器各向前推进一步:

```
template<typename I, typename O>
    requires(Readable(I) && Iterator(I) &&
        Writable(O) && Iterator(O) &&
        ValueType(I) == ValueType(O))
void copy_step(I& f_i, O& f_o)
{
    // 前条件: source(f_i) 和 sink(f_o) 有定义
    sink(f_o) = source(f_i);
    f_i = successor(f_i);
    f_o = successor(f_o);
}
```

拷贝算法的一般形式是反复执行一种基本的拷贝步骤, 直至某个终止条件满足. 举例说, 下面的 copy 将一个半开的有界范围拷贝到一个输出范围, 相应输出范围由该范围的第一个迭代器描述:

```
template<typename I, typename O>
    requires(Readable(I) && Iterator(I) &&
        Writable(O) && Iterator(O) &&
        ValueType(I) == ValueType(O))
O copy(I f_i, I l_i, O f_o)
{
    // 前条件: not_overlapped_forward(f_i, l_i, f_o, f_o + (l_i - f_i))
    while (f_i != l_i) copy_step(f_i, f_o);
    return f_o;
}
```

copy 返回输出范围的极限, 这是因为调用方原来可能不知道它. 输出迭代器类型可能不允许多遍遍历, 在这种情况下, 如果不返回这个极限, 后面可能就无法再找到它了.

copy 的后条件是输出范围里的值序列等于输入范围里原来的值序列. 为满足这一后条件, 前条件必须保证输入和输出范围分别可读和可写; 输出范围必须足够大; 还有, 如果输入范围和输出范围重叠, 那么不会出现通过一个输出迭代器写之后再通过与它别名的某输入迭代器去读的情况. 这些条件可以借助性质 not_overlapped_forward 的帮助进行形式化. 一个可读范围和一个可写范围没有重叠的前程 (overlapped forward), 条件是, 如果任何别名迭代器出现在输入范围里, 其在输入范围里的索引都不超过其在输出范围里的索引:

property(I : *Readable*, O : *Writable*)
  requires(*Iterator*(I) ∧ *Iterator*(O))
not_overlapped_forward : I × I × O × O
  $(f_i, l_i, f_o, l_o) \mapsto$
    readable_bounded_range(f_i, l_i) ∧
    writable_bounded_range(f_o, l_o) ∧
    $(\forall k_i \in [f_i, l_i))(\forall k_o \in [f_o, l_o))$
      aliased(k_o, k_i) ⇒ $k_i - f_i \leqslant k_o - f_o$

有时输入范围和输出范围的规模可能不同:

```
template<typename I, typename O>
    requires(Readable(I) && Iterator(I) &&
        Writable(O) && Iterator(O) &&
        ValueType(I) == ValueType(O))
pair<I, O> copy_bounded(I f_i, I l_i, O f_o, O l_o)
{
    // 前条件: not_overlapped_forward(f_i, l_i, f_o, l_o)
    while (f_i != l_i && f_o != l_o) copy_step(f_i, f_o);
    return pair<I, O>(f_i, f_o);
}
```

如果调用方同时知道两个范围的结束位置, 返回这样的一对迭代器, 就使调用方可以确定哪个范围较小, 以及较大范围里的拷贝在哪里结束. 与 copy 比较一下, 可以看到输出的前条件被弱化了: 现在输出范围可以比输入范围短.

有人甚至会说最弱的前条件应该是

$$\text{not\_overlapped\_forward}(f_i, f_i + n, f_o, f_o + n)$$

其中 $n = \min(l_i - f_i, l_o - f_o)$.

下面辅助机器处理计数范围的终止条件:

```
template<typename N>
    requires(Integer(N))
bool count_down(N& n)
{
    // 前条件: n ≥ 0
    if (zero(n)) return false;
    n = predecessor(n);
    return true;
}
```

下面的 copy_n 算法将一个半开的计数范围拷贝到一个输出范围, 相应输出范围由该范围的第一个迭代器描述:

```
template<typename I, typename O, typename N>
    requires(Readable(I) && Iterator(I) &&
        Writable(O) && Iterator(O) &&
        ValueType(I) == ValueType(O) &&
        Integer(N))
pair<I, O> copy_n(I f_i, N n, O f_o)
{
    // 前条件: not_overlapped_forward(f_i, f_{i+n}, f_o, f_{o+n})
    while (count_down(n)) copy_step(f_i, f_o);
    return pair<I, O>(f_i, f_o);
}
```

对两个计数范围做 copy_bounded 的效果, 也可以通过以两者之中较小的范围的规模调用 copy_n 的方式获得.

当范围之间有重叠的前程时, 如果所用的迭代器建模 *BidirectionalIterator* 并因此允许反向走, 那么仍然可以拷贝. 这一情况提示了下面的机器:

```
template<typename I, typename O>
    requires(Readable(I) && BidirectionalIterator(I) &&
        Writable(O) && BidirectionalIterator(O) &&
        ValueType(I) == ValueType(O))
void copy_backward_step(I& l_i, O& l_o)
{
    // 前条件: source(predecessor(l_i)) 和 sink(predecessor(l_o)) 有定义
    l_i = predecessor(l_i);
    l_o = predecessor(l_o);
    sink(l_o) = source(l_i);
}
```

由于这里处理的是半开范围, 并需要从它的极限位置开始, 因此需要在拷贝之前先将迭代器退一步, 这样就得到了 copy_backward:

```
template<typename I, typename O>
    requires(Readable(I) && BidirectionalIterator(I) &&
        Writable(O) && BidirectionalIterator(O) &&
        ValueType(I) == ValueType(O))
O copy_backward(I f_i, I l_i, O l_o)
{
    // 前条件: not_overlapped_backward(f_i, l_i, l_o - (l_i - f_i), l_o)
    while (f_i != l_i) copy_backward_step(l_i, l_o);
    return l_o;
}
```

copy_backward_n 与此类似.

copy_backward 的后条件类似于 copy, 可以利用性质 not_overlapped_backward 对它进行形式化. 一个读范围和一个写范围没有重叠的后程 (overlapped backward), 如果任何别名迭代器出现在输入范围里的从极限开始计算的索引, 都不超出它在输出范围里从极限开始计算的索引:

property(I : *Readable*, O : *Writable*)

requires(*Iterator*(I) ∧ *Iterator*(O))

not_overlapped_backward : I × I × O × O

$(f_i, l_i, f_o, l_o) \mapsto$

readable_bounded_range(f_i, l_i) ∧

writable_bounded_range(f_o, l_o) ∧

$(\forall k_i \in [f_i, l_i))(\forall k_o \in [f_o, l_o))$

aliased$(k_o, k_i) \Rightarrow l_i - k_i \leqslant l_o - k_o$

如果两个范围里的迭代器类型都建模 *BidirectionalIterator*, 就可以像反转输入范围的方向一样反转输出范围的方向. 下面是这样一部机器, 它能一边在输出范围里反向走, 一边也在输入范围里反向走:

```
template<typename I, typename O>
    requires(Readable(I) && BidirectionalIterator(I) &&
             Writable(O) && Iterator(O) &&
             ValueType(I) == ValueType(O))
void reverse_copy_step(I& l_i, O& f_o)
{
    // 前条件: source(predecessor(l_i)) 和 sink(f_o) 有定义
    l_i = predecessor(l_i);
    sink(f_o) = source(l_i);
    f_o = successor(f_o);
}

template<typename I, typename O>
    requires(Readable(I) && Iterator(I) &&
             Writable(O) && BidirectionalIterator(O) &&
             ValueType(I) == ValueType(O))
void reverse_copy_backward_step(I& f_i, O& l_o)
{
    // 前条件: source(f_i) 和 sink(predecessor(l_o)) 有定义
    l_o = predecessor(l_o);
```

```
    sink(l_o) = source(f_i);
    f_i = successor(f_i);
}
```

这样就得到了下面的算法:

```
template<typename I, typename O>
    requires(Readable(I) && BidirectionalIterator(I) &&
        Writable(O) && Iterator(O) &&
        ValueType(I) == ValueType(O))
O reverse_copy(I f_i, I l_i, O f_o)
{
    // 前条件: not_overlapped(f_i, l_i, f_o, f_o + (l_i - f_i))
    while (f_i != l_i) reverse_copy_step(l_i, f_o);
    return f_o;
}
```

```
template<typename I, typename O>
    requires(Readable(I) && Iterator(I) &&
        Writable(O) && BidirectionalIterator(O) &&
        ValueType(I) == ValueType(O))
O reverse_copy_backward(I f_i, I l_i, O l_o)
{
    // 前条件: not_overlapped(f_i, l_i, l_o - (l_i - f_i), l_o)
    while (f_i != l_i) reverse_copy_backward_step(f_i, l_o);
    return l_o;
}
```

reverse_copy_n 与 reverse_copy_backward_n 类似.

reverse_copy 和 reverse_copy_backward 的后条件是: 输出范围是输入范围里原有的值序列的反转拷贝. 实用 (但不是最弱) 的前条件是输入和输出范围不重叠, 这一条件可以借助性质 not_overlapped 进行形式化. 一个可读范围和一个可写范围不重叠 (overlapped) 的条件是它们没有共同的迭代器:

property(I : *Readable*, O : *Writable*)
  requires(Iterator(I) ∧ Iterator(O))
not_overlapped : I × I × O × O
  $(f_i, l_i, f_o, l_o) \mapsto$
    readable_bounded_range(f_i, l_i) ∧
    writable_bounded_range(f_o, l_o) ∧
    $(\forall k_i \in [f_i, l_i))\ (\forall k_o \in [f_o, l_o))\ \neg$aliased($k_o, k_i$)

练习 9.1 请找出 reverse_copy 和 reverse_copy_backward 的最弱前条件.

引进 copy_backward 和 copy 的主要原因是要处理在任何方向上重叠的范围, 而引进 reverse_copy_backward 和 reverse_copy, 则是为了能在针对迭代器的要求上取得最大的灵活性.

9.3 基于谓词的拷贝

前面给出的算法都把输入范围里的所有对象拷贝到输出范围, 它们的后条件也不依赖于任何迭代器的值. 本节的几个算法都要求一个谓词参数, 通过它来控制拷贝工作的进行.

例如, 让拷贝步骤以一个一元谓词为条件, 就得到了 copy_select:

```
template<typename I, typename O, typename P>
    requires(Readable(I) && Iterator(I) &&
        Writable(O) && Iterator(O) &&
        ValueType(I) == ValueType(O) &&
        UnaryPredicate(P) && I == Domain(P))
O copy_select(I f_i, I l_i, O f_t, P p)
{
    // 前条件: not_overlapped_forward(f_i, l_i, f_t, f_t + n_t)
    // 其中的 n_t 是满足 p 的迭代器数目的一个上界
    while (f_i != l_i)
        if (p(f_i)) copy_step(f_i, f_t);
        else f_i = successor(f_i);
```

```
    return f_t;
}
```

n_t 的最坏情况是 $l_i - f_i$; 上下文可能保证它取更小的值.

更常见的情况是谓词不作用于迭代器, 而是作用于它们的值:

```
template<typename I, typename O, typename P>
    requires(Readable(I) && Iterator(I) &&
        Writable(O) && Iterator(O) &&
        ValueType(I) == ValueType(O) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
O copy_if(I f_i, I l_i, O f_t, P p)
{
    // 前条件: 与 copy_select 相同
    predicate_source<I, P> ps(p);
    return copy_select(f_i, l_i, f_t, ps);
}
```

第 8 章里给出了在迭代器的链接范围上操作的 split_linked 和 combine_linked_nonempty. 也存在与之类似的拷贝算法:

```
template<typename I, typename O_f, typename O_t, typename P>
    requires(Readable(I) && Iterator(I) &&
        Writable(O_f) && Iterator(O_f) &&
        Writable(O_t) && Iterator(O_t) &&
        ValueType(I) == ValueType(O_f) &&
        ValueType(I) == ValueType(O_t) &&
        UnaryPredicate(P) && I == Domain(P))
pair<O_f, O_t> split_copy(I f_i, I l_i, O_f f_f, O_t f_t,
                          P p)
{
    // 前条件: 见下
    while (f_i != l_i)
        if (p(f_i)) copy_step(f_i, f_t);
```

```
        else          copy_step(f_i, f_f);
    return pair<O_f, O_t>(f_f, f_t);
}
```

练习 9.2 请写出 split_copy 的后条件.

为了满足其后条件, 对 split_copy 的调用必须保证两个输出范围绝不重叠. 实际上也可以允许其中一个输出范围与输入范围重叠, 但要保证它们没有重叠的前程 (overlapped forward). 这样就得到了下面前条件:

$$\begin{aligned}&\mathsf{not\_write\_overlapped}(f_f, n_f, f_t, n_t) \wedge \\ &\quad((\mathsf{not\_overlapped\_forward}(f_i, l_i, f_f, f_f + n_f) \wedge \mathsf{not\_overlapped}(f_i, l_i, f_t, l_t)) \vee \\ &\quad\ (\mathsf{not\_overlapped\_forward}(f_i, l_i, f_t, f_t + n_t) \wedge \mathsf{not\_overlapped}(f_i, l_i, f_f, l_f)))\end{aligned}$$

其中的 n_f 和 n_t 分别为不满足和满足 p 的迭代器数目的上界.

性质 not_write_overlapped 的定义依赖于写别名 (write aliasing) 的概念: 这一说法是指两个可写对象 x 和 y, 对它们 sink(x) 和 sink(y) 都有定义, 而且任何可以看到写 x 的效果的观察者也能看到写 y 的效果:

$$\begin{aligned}&\textbf{property}(\mathsf{T} : \mathit{Writable}, \mathsf{U} : \mathit{Writable}) \\ &\quad\textbf{requires}(\mathsf{ValueType}(\mathsf{T}) = \mathsf{ValueType}(\mathsf{U})) \\ &\mathsf{write\_aliased} : \mathsf{T} \times \mathsf{U} \\ &\quad(x, y) \mapsto \mathsf{sink}(x) \text{ 有定义} \wedge \mathsf{sink}(y) \text{ 有定义} \wedge \\ &\qquad(\forall V \in \mathit{Readable})\,(\forall v \in V)\,\mathsf{aliased}(x, v) \Leftrightarrow \mathsf{aliased}(y, v)\end{aligned}$$

这样就得到了无写重叠 (not write overlapped) 的定义, 也就是说, 两个可写范围没有公共的别名 sink:

$$\begin{aligned}&\textbf{property}(O_0 : \mathit{Writable}, O_1 : \mathit{Writable}) \\ &\quad\textbf{requires}(\mathsf{Iterator}(O_0) \wedge \mathsf{Iterator}(O_1)) \\ &\mathsf{not\_write\_overlapped} : O_0 \times O_0 \times O_1 \times O_1 \\ &\quad(f_0, l_0, f_1, l_1) \mapsto \\ &\qquad\mathsf{writable\_bounded\_range}(f_0, l_0) \wedge \\ &\qquad\mathsf{writable\_bounded\_range}(f_1, l_1) \wedge\end{aligned}$$

$$(\forall k_0 \in [f_0, l_0))(\forall k_1 \in [f_1, l_1))\neg \text{write\_aliased}(k_0, k_1)$$

与 select_copy 的情况类似, 在 split_copy 的最常见情况中, 谓词并不作用于迭代器本身, 而是作用于它们的值:[2]

```
template<typename I, typename O_f, typename O_t, typename P>
    requires(Readable(I) && Iterator(I) &&
        Writable(O_f) && Iterator(O_f) &&
        Writable(O_t) && Iterator(O_t) &&
        ValueType(I) == ValueType(O_f) &&
        ValueType(I) == ValueType(O_t) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
pair<O_f, O_t> partition_copy(I f_i, I l_i, O_f f_f, O_t f_t,
                              P p)
{
    // 前条件: 与 split_copy 相同
    predicate_source<I, P> ps(p);
    return split_copy(f_i, l_i, f_f, f_t, ps);
}
```

两个输出范围里的值都保持原输入范围里的相对顺序; partition_copy_n 的情况与此类似.

combine_copy 的代码同样很简单:

```
template<typename I0, typename I1, typename O, typename R>
    requires(Readable(I0) && Iterator(I0) &&
        Readable(I1) && Iterator(I1) &&
        Writable(O) && Iterator(O) &&
        BinaryPredicate(R) &&
        ValueType(I0) == ValueType(O) &&
        ValueType(I1) == ValueType(O) &&
        I0 == InputType(R, 1) && I1 == InputType(R, 0))
O combine_copy(I0 f_i0, I0 l_i0, I1 f_i1, I1 l_i1, O f_o, R r)
```

2. 这一接口是 T. K. Lakshman 的建议.

```
{
    // 前条件: 见下
    while (f_i0 != l_i0 && f_i1 != l_i1)
        if (r(f_i1, f_i0)) copy_step(f_i1, f_o);
        else               copy_step(f_i0, f_o);
    return copy(f_i1, l_i1, copy(f_i0, l_i0, f_o));
}
```

对于 combine_copy, 输入范围之间的读重叠是可以接受的. 进一步说, 也允许一个输入范围与输出范围重叠, 但这种重叠不能在向前的方向, 而且在反方向的偏移量至少要有另一个输入范围那么大. 这些要求在 combine_copy 的前条件里用 backward_offset 性质描述:

$$(\mathsf{backward\_offset}(f_{i0}, l_{i0}, f_o, l_o, l_{i1} - f_{i1}) \wedge \mathsf{not\_overlapped}(f_{i1}, l_{i1}, f_o, l_o)) \vee$$
$$(\mathsf{backward\_offset}(f_{i1}, l_{i1}, f_o, l_o, l_{i0} - f_{i0}) \wedge \mathsf{not\_overlapped}(f_{i0}, l_{i0}, f_o, l_o))$$

其中的 $l_o = f_o + (l_{i0} - f_{i0}) + (l_{i1} - f_{i1})$ 是输出范围的极限.

一个可读范围、一个可写范围和一个偏移量 $n \geqslant 0$ 满足 backward_offset 性质的条件是: 如果在输入范围的一个索引处出现任何别名的迭代器, 对其增加 n, 也不会超过输出范围的索引:

$$\begin{aligned}
&\textbf{property}(I : \mathit{Readable}, O : \mathit{Writable}, N : \mathit{Integer})\\
&\quad \textbf{requires}(\mathit{Iterator}(I) \wedge \mathit{Iterator}(O))\\
&\mathsf{backward_offset} : I \times I \times O \times O \times N\\
&\quad (f_i, l_i, f_o, l_o, n) \mapsto\\
&\qquad \mathsf{readable_bounded_range}(f_i, l_i) \wedge\\
&\qquad n \geqslant 0 \wedge\\
&\qquad \mathsf{writable_bounded_range}(f_o, l_o) \wedge\\
&\qquad (\forall k_i \in [f_i, l_i))(\forall k_o \in [f_o, l_o))\\
&\qquad\quad \mathsf{aliased}(k_o, k_i) \Rightarrow k_i - f_i + n \leqslant k_o - f_o
\end{aligned}$$

请注意, $\mathsf{not\_overlapped\_forward}(f_i, l_i, f_o, l_o) = \mathsf{backward\_offset}(f_i, l_i, f_o, l_o, 0)$.

练习 9.3 请写出 combine_copy 的后条件, 并证明只要前条件成立, 后条件也必然满足.

combine_copy_backward 的情况类似. 要保证同样的后条件成立, if 分支的顺序必须与 combine_copy 里 if 分支的顺序相反:

```
template<typename I0, typename I1, typename O, typename R>
    requires(Readable(I0) && BidirectionalIterator(I0) &&
        Readable(I1) && BidirectionalIterator(I1) &&
        Writable(O) && BidirectionalIterator(O) &&
        BinaryPredicate(R) &&
        ValueType(I0) == ValueType(O) &&
        ValueType(I1) == ValueType(O) &&
        I0 == InputType(R, 1) && I1 == InputType(R, 0))
O combine_copy_backward(I0 f_i0, I0 l_i0, I1 f_i1, I1 l_i1,
                        O l_o, R r)
{
    // 前条件: 见下
    while (f_i0 != l_i0 && f_i1 != l_i1) {
        if (r(predecessor(l_i1), predecessor(l_i0)))
            copy_backward_step(l_i0, l_o);
        else
            copy_backward_step(l_i1, l_o);
    }
    return copy_backward(f_i0, l_i0,
                         copy_backward(f_i1, l_i1, l_o));
}
```

combine_copy_backward 的前条件是

$$(\text{forward\_offset}(f_{i_0}, l_{i_0}, f_o, l_o, l_{i_1} - f_{i_1}) \wedge \text{not\_overlapped}(f_{i_1}, l_{i_1}, f_o, l_o)) \vee$$
$$(\text{forward\_offset}(f_{i_1}, l_{i_1}, f_o, l_o, l_{i_0} - f_{i_0}) \wedge \text{not\_overlapped}(f_{i_0}, l_{i_0}, f_o, l_o))$$

其中 $f_o = l_o - (l_{i_0} - f_{i_0}) + (l_{i_1} - f_{i_1})$ 是输出范围的第一个迭代器.

一个可读范围、一个可写范围和一个偏移量 $n \geqslant 0$ 满足性质 forward_offset 的条件是: 如果任何别名迭代器出现在从输入范围的极限开始算的某个索引, 将其增加 n, 也不会超过相应别名迭代器从输出范围的极限开始算的索引:

property(I : *Readable*, O : *Writable*, N : *Integer*)

requires(*Iterator*(I) ∧ *Iterator*(O))

forward_offset : I × I × O × O × N

$(f_i, l_i, f_o, l_o, n) \mapsto$

readable_bounded_range(f_i, l_i) ∧

$n \geqslant 0 \wedge$

writable_bounded_range(f_o, l_o) ∧

$(\forall k_i \in [f_i, l_i))(\forall k_o \in [f_o, l_o))$

aliased$(k_o, k_i) \Rightarrow l_i - k_i + n \leqslant l_o - k_o$

注意, not_overlapped_backward(f_i, l_i, f_o, l_o) = forward_offset$(f_i, l_i, f_o, l_o, 0)$.

练习 9.4 请写出 combine_copy_backward 的后条件, 并证明只要其前条件成立, 后条件也必然满足.

把值类型上的一个弱序送给前向或反向组合拷贝算法, 它们都能归并递增的范围:

```
template<typename I0, typename I1, typename O, typename R>
    requires(Readable(I0) && Iterator(I0) &&
        Readable(I1) && Iterator(I1) &&
        Writable(O) && Iterator(O) &&
        Relation(R) &&
        ValueType(I0) == ValueType(O) &&
        ValueType(I1) == ValueType(O) &&
        ValueType(I0) == Domain(R))
O merge_copy(I0 f_i0, I0 l_i0, I1 f_i1, I1 l_i1, O f_o, R r)
{
    // 前条件: 除 combine_copy 的前条件, 还要求
    //     weak_ordering(r) ∧
    //     increasing_range(f_i0, l_i0, r) ∧ increasing_range(f_i1, l_i1, r)
    relation_source<I1, I0, R> rs(r);
    return combine_copy(f_i0, l_i0, f_i1, l_i1, f_o, rs);
}
```

```
template<typename I0, typename I1, typename O, typename R>
    requires(Readable(I0) && BidirectionalIterator(I0) &&
        Readable(I1) && BidirectionalIterator(I1) &&
        Writable(O) && BidirectionalIterator(O) &&
        Relation(R) &&
        ValueType(I0) == ValueType(O) &&
        ValueType(I1) == ValueType(O) &&
        ValueType(I0) == Domain(R))
O merge_copy_backward(I0 f_i0, I0 l_i0, I1 f_i1, I1 l_i1, O l_o,
                      R r)
{
    // 前条件: 除 combine_copy_backward 的前条件, 还要求
    //     weak_ordering(r) ∧
    //     increasing_range(f_i0, l_i0, r) ∧ increasing_range(f_i1, l_i1, r)
    relation_source<I1, I0, R> rs(r);
    return combine_copy_backward(f_i0, l_i0, f_i1, l_i1, l_o,
                                 rs);
}
```

练习 9.5 实现 combine_copy_n 和 combine_copy_backward_n, 给出适当返回值.

引理 9.1 如果两个输入范围规模分别为 n_0 和 n_1, 在最坏情况下 merge_copy 和 merge_copy_backward 将执行 $n_0 + n_1$ 次赋值及 $n_0 + n_1 - 1$ 次比较.

练习 9.6 请确定最好的和平均的比较次数.

项目 9.1 现代计算系统都为内存拷贝提供了高度优化的库过程; 例如 memmove 和 memcpy, 其中使用了本书中没有讨论的优化技术. 请研究你的平台提供的过程, 设法确定它们采用的技术 (例如, 循环展开和软件流水线), 并设计出抽象过程来尽可能地表达这些技术. 每种技术需要哪些类型需求和前条件? 哪些语言扩充使编译器能拥有采纳这些技术的全部灵活性?

9.4 范围的交换

程序里除了经常需要将一个范围拷贝到另一个范围外, 有时也很需要交换 (swap) 两个同样大小的范围: 一一对换两个范围里处于同样位置的各对对象的值. 交换算法很像前面的拷贝算法, 只是其中的赋值用一个过程取代, 该过程交换由两个可变动迭代器指着的对象的值:

```
template<typename I0, typename I1>
    requires(Mutable(I0) && Mutable(I1) &&
        ValueType(I0) == ValueType(I1))
void exchange_values(I0 x, I1 y)
{
    // 前条件: deref(x) 和 deref(y) 有定义
    ValueType(I0) t = source(x);
            sink(x) = source(y);
            sink(y) = t;
}
```

练习 9.7 exchange_values 的后条件是什么?

引理 9.2 exchange_values(i, j) 和 exchange_values(j, i) 的效果等价.

我们可能希望 exchange_values 的实现只是交换两个对象值, 而不实际构造或销毁任何对象, 这样它的代价就不会随着被交换对象拥有的资源量的增加而增加. 第 12 章里将通过基础类型 (underlying type) 的概念达到这一目标.

就像做拷贝一样, 现在也从几个机器出发构造各种交换算法, 这些机器都以两个迭代器的引用作为参数, 完成交换工作并更新有关的迭代器. 下面机器在交换了两个对象之后推进这两个迭代器:

```
template<typename I0, typename I1>
    requires(Mutable(I0) && ForwardIterator(I0) &&
        Mutable(I1) && ForwardIterator(I1) &&
        ValueType(I0) == ValueType(I1))
void swap_step(I0& f0, I1& f1)
{
```

```
    // 前条件: deref(f0) 和 deref(f1) 有定义
    exchange_values(f0, f1);
    f0 = successor(f0);
    f1 = successor(f1);
}
```

这样就得到第一个算法, 它与 copy 类似:

```
template<typename I0, typename I1>
    requires(Mutable(I0) && ForwardIterator(I0) &&
        Mutable(I1) && ForwardIterator(I1) &&
        ValueType(I0) == ValueType(I1))
I1 swap_ranges(I0 f0, I0 l0, I1 f1)
{
    // 前条件: mutable_bounded_range(f0, l0)
    // 前条件: mutable_counted_range(f1, l0 - f0)
    while (f0 != l0) swap_step(f0, f1);
    return f1;
}
```

第二个算法与 copy_bounded 类似:

```
template<typename I0, typename I1>
    requires(Mutable(I0) && ForwardIterator(I0) &&
        Mutable(I1) && ForwardIterator(I1) &&
        ValueType(I0) == ValueType(I1))
pair<I0, I1> swap_ranges_bounded(I0 f0, I0 l0, I1 f1, I1 l1)
{
    // 前条件: mutable_bounded_range(f0, l0)
    // 前条件: mutable_bounded_range(f1, l1)
    while (f0 != l0 && f1 != l1) swap_step(f0, f1);
    return pair<I0, I1>(f0, f1);
}
```

第三个算法与 copy_n 类似:

```
template<typename I0, typename I1, typename N>
    requires(Mutable(I0) && ForwardIterator(I0) &&
        Mutable(I1) && ForwardIterator(I1) &&
        ValueType(I0) == ValueType(I1) &&
        Integer(N))
pair<I0, I1> swap_ranges_n(I0 f0, I1 f1, N n)
{
    // 前条件: mutable_counted_range(f0, n)
    // 前条件: mutable_counted_range(f1, n)
    while (count_down(n)) swap_step(f0, f1);
    return pair<I0, I1>(f0, f1);
}
```

如果送给范围交换算法的范围不重叠, 情况很清楚, 上面算法的效果就是交换了对应位置的各对对象的值. 下一章将推导出存在重叠情况时的后条件.

反向拷贝的结果也是得到了一份拷贝, 其中的位置与原范围相比是反过来的. 反向交换的情况与此类似. 要做反向交换, 需要有另一机器, 它向后推第一个范围, 同时向前推第二个范围:

```
template<typename I0, typename I1>
    requires(Mutable(I0) && BidirectionalIterator(I0) &&
        Mutable(I1) && ForwardIterator(I1) &&
        ValueType(I0) == ValueType(I1))
void reverse_swap_step(I0& l0, I1& f1)
{
    // 前条件: deref(predecessor(l0)) 和 deref(f1) 有定义
    l0 = predecessor(l0);
    exchange_values(l0, f1);
    f1 = successor(f1);
}
```

由于 exchange_values 的对称性, 只要有一个迭代器的类型是双向的, 就可以用 reverse_swap_ranges, 不需要再写另一个反向的版本:

```
template<typename I0, typename I1>
    requires(Mutable(I0) && BidirectionalIterator(I0) &&
        Mutable(I1) && ForwardIterator(I1) &&
        ValueType(I0) == ValueType(I1))
I1 reverse_swap_ranges(I0 f0, I0 l0, I1 f1)
{
    // 前条件: mutable_bounded_range(f0, l0)
    // 前条件: mutable_counted_range(f1, l0 - f0)
    while (f0 != l0) reverse_swap_step(l0, f1);
    return f1;
}

template<typename I0, typename I1>
    requires(Mutable(I0) && BidirectionalIterator(I0) &&
        Mutable(I1) && ForwardIterator(I1) &&
        ValueType(I0) == ValueType(I1))
pair<I0, I1>reverse_swap_ranges_bounded(I0 f0, I0 l0,
                                        I1 f1, I1 l1)
{
    // 前条件: mutable_bounded_range(f0, l0)
    // 前条件: mutable_bounded_range(f1, l1)
    while (f0 != l0 && f1 != l1)
        reverse_swap_step(l0, f1);
    return pair<I0, I1>(l0, f1);
}

template<typename I0, typename I1, typename N>
    requires(Mutable(I0) && BidirectionalIterator(I0) &&
        Mutable(I1) && ForwardIterator(I1) &&
        ValueType(I0) == ValueType(I1) &&
        Integer(N))
pair<I0, I1> reverse_swap_ranges_n(I0 l0, I1 f1, N n)
```

```
{
    // 前条件: mutable_counted_range(l_0 − n, n)
    // 前条件: mutable_counted_range(f_1, n)
    while (count_down(n)) reverse_swap_step(l0, f1);
    return pair<I0, I1>(l0, f1);
}
```

9.5 总结

通过扩展迭代器类型使之支持 sink，带来了可写性和可变动性. sink 的公理很简单, 但如果有别名和并发更新 (本书中没有讨论), 命令式程序设计也会变得非常复杂. 特别是, 在定义前条件使之能处理不同迭代器类型的别名时, 需要特别小心. 各种拷贝算法很简单, 功能强大, 应用也非常广泛. 从简单的机器出发组合出这些算法, 能帮助我们把这些算法组织起来, 揭示出它们的共同点, 也能说明它们之间的差异. 使用值交换而不是赋值, 就得到了一族类似的但可能稍少的范围交换算法, 它们都非常有用.

第 10 章

重整

本章首先介绍置换的概念，还要介绍一批算法的分类体系. 这些算法称为重整，它们排列一个范围里的元素以满足某个给定的后条件. 这里要为双向和随机访问迭代器提出一些迭代式的反转算法，为前向迭代器提出一个采用分治方式的反转算法. 还将展示如何对该分治算法做一些变换，使之在能使用额外内存的情况下运行更快. 本章还要描述针对不同迭代器类型的三个轮换算法，其中的轮换操作交换两个相邻的范围，这两个范围可以不一样大. 在本章最后还将讨论如何基于不同算法的需求，将它们打包为在编译时选择的算法.

10.1 置换

变换 f 是内变换 (into transformation)，如果对其定义空间里任意的 x，都存在定义空间里的 y 使 $y = f(x)$. f 是满变换 (onto transformation)，如果对其定义空间里任意的 y，都存在定义空间里的某个 x 使 $y = f(x)$. f 是一一变换 (one-to-one transformation)，如果对其定义空间里任意的 x, x'，$f(x) = f(x') \Rightarrow x = x'$.

引理 10.1 有穷定义空间上的 f 是满变换，当且仅当它既是内变换也是一一变换.

练习 10.1 请找出自然数上的一个变换，它既是内变换又是满变换，但却不是一一变换；以及一个既内又一一的，但却非满的变换.

满足 $f(x) = x$ 的元素 x 称为变换 f 的不动点 (fixed point). 恒等变换 (identity transformation) 是以其定义空间里的每个元素为不动点的变换. 下面将用 $\mathsf{identity}_S$ 表示集合 S 上的恒等变换.

置换 (permutation) 就是有限定义空间上的满变换. 下面是 $[0,6)$ 上的一个置换实例:

$$p(0) = 5$$
$$p(1) = 2$$
$$p(2) = 4$$
$$p(3) = 3$$
$$p(4) = 1$$
$$p(5) = 0$$

如果 p 和 q 是集合 S 上的两个置换, 它们的**复合** (composition), 记为 $q \circ p$, 将 $x \in S$ 变到 $q(p(x))$.

引理 10.2 置换的复合还是置换.

引理 10.3 置换的复合运算是可结合的.

引理 10.4 对于集合 S 上的每个置换 p, 都存在其**逆置换** (inverse of permutation) p^{-1} 使得 $p^{-1} \circ p = p \circ p^{-1} = \mathsf{identity}_S$.

一个集合上的所有置换在复合下形成一个群 (permutation group).

引理 10.5 任何有穷群都是其元素的置换群的一个子群, 这一子群里的一个置换通过用群中的一个元素乘以其他各元素生成.

例如, 模 5 乘法群有下面的乘法表:

×	**1**	**2**	**3**	**4**
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

这一乘法表里的每一行或一列都是一个置换. 易见并不是 4 个元素上的 $4! = 24$ 个置换都在这个群里, 因此模 5 的**乘法群**是四个元素的置换群的一个真子群.

一个**环路** (cycle) 就是置换中的一条环形轨道. **平凡环路** (trivial cycle) 指规模为 1 的环路; 每个平凡环路中的元素都是相应置换的不动点. 包含恰好一个

非平凡环路的置换称为一个**循环置换** (cyclic permutation). **对换** (transposition) 是具有规模为 2 的环路的循环置换.

引理 10.6 置换中的每个元素属于唯一的一个环路.

引理 10.7 n 元集合上的任一置换包含 $k \leqslant n$ 个环路.

引理 10.8 不相交的循环置换可以交换.

练习 10.2 请给出一个例子, 其中有两个相交的循环置换, 它们不可交换.

引理 10.9 每个置换都能表示为与它的环路对应的所有循环置换的乘积.

引理 10.10 一个置换的逆是其各个环路的逆的乘积.

引理 10.11 每个循环置换都是一些对换的乘积.

引理 10.12 每个置换都是一些对换的乘积.

规模为 n 的**有穷集** (finite set) S 是一个集合, 对它存在一对函数

$$\mathrm{choose}_S : [0, n) \to S$$

$$\mathrm{index}_S : S \to [0, n)$$

满足

$$\mathrm{choose}_S(\mathrm{index}_S(x)) = x$$

$$\mathrm{index}_S(\mathrm{choose}_S(i)) = i$$

换句话说, S 可与自然数的一个区间建立一一对应.

如果 p 是规模为 n 的有穷集 S 的一个置换, 那么存在 $[0, n)$ 上对应的**索引置换** p', 定义如下

$$p'(i) = \mathrm{index}_S(p(\mathrm{choose}_S(i)))$$

引理 10.13 $p(x) = \mathrm{choose}_S(p'(\mathrm{index}_S(x)))$

下面经常用对应的索引置换来定义集合上的置换.

10.2 重整

一个**重整** (rearrangement) 是一个算法, 它把一个输入范围里的元素拷贝到一个输出范围, 使这一对输入和输出范围之间的索引映射是一个置换. 本章研究**基于位置的** (position-based) 重整, 其中一个值的目标位置只依赖于它原来的位置而不依赖于它的值. 下一章将研究**基于谓词的** (predicate-based) 重整, 在那里一个值的目标位置只依赖于将一个谓词作用于该值的结果; 还有**基于序的** (ordering-based) 重整, 其中一个值的目标位置仅依赖于值之间的顺序关系.

第 8 章研究了链接的重整, 例如 reverse_linked, 其中通过修改链接完成重整工作. 第 9 章研究了拷贝重整, 例如 copy 和 reverse_copy. 本章和下一章将研究**变动型重整**, 其中的输入和输出是同一个范围.

每一个变动型重整都对应于两个置换: 一个**去置换** (to permutation), 它把一个迭代器 i 映射到指向原本在 i 的元素的目标位置的迭代器; 还有一个**回置换** (from permutation), 它将迭代器 i 映射到指向被移到了 i 的元素的原来位置的那个迭代器.

引理 10.14 一个重排的去置换和回置换互逆.

如果去置换已知, 那么就可以用下面算法重整一个环路:

```
template<typename I, typename F>
    requires(Mutable(I) && Transformation(F) && I == Domain(F))
void cycle_to(I i, F f)
{
    // 前条件: i 在 f 下的轨道是环路
    // 前条件: (∀n ∈ ℕ) deref(f^n(i)) 有定义
    I k = f(i);
    while (k != i) {
        exchange_values(i, k);
        k = f(k);
    }
}
```

在执行 cycle_to(i, f) 之后, 对于 i 在 f 下的轨道上的所有迭代器 j, 都有 source(f(j)) 的值与 source(j) 的原值相等. 对规模为 n 的环路, 这一调用将执行 $3(n-1)$ 次赋值.

练习 10.3 请实现 cycle_to 的一个版本, 它只需做 $2n-1$ 次赋值.

如果回置换已知, 可以用下面算法重整一个环路:

```
template<typename I, typename F>
    requires(Mutable(I) && Transformation(F) && I == Domain(F))
void cycle_from(I i, F f)
{
    // 前条件: i 在 f 下的轨道是环路
    // 前条件: (∀n ∈ ℕ) deref(f^n(i)) 有定义
    ValueType(I) tmp = source(i);
    I j = i;
    I k = f(i);
    while (k != i) {
        sink(j) = source(k);
        j = k;
        k = f(k);
    }
    sink(j) = tmp;
}
```

在执行了 cycle_from(i, f) 之后, 对于 i 在 f 的轨道上的所有迭代器 j, source(j) 的值和 source(f(j)) 的原值相等. 这个调用执行 $n+1$ 次赋值, 而如果用 exchange_values 实现它, 就需要执行 $3(n-1)$ 次赋值. 可以看到, 这里只要求类型 I 上的可修改性. 而且不需要任何遍历函数, 因为变换 f 执行了有关的遍历. 除了回置换外, 利用 cycle_from 实现修改型重整还需要有一种方式取得每个环路的代表元. 在一些实际情况下, 环路的结构及其代表元是已知的.

练习 10.4 请实现一个算法, 它完成一个索引迭代器的范围里的一次任意重整. 请用一个包含 n 个布尔值的数组来标记已经重新置位的元素, 扫描该数组找

未标记的元素, 通过这种方法确定下一个环的代表元.

练习 10.5 假设迭代器有一个全序, 请设计一个算法, 它只需常量空间就可以确定某迭代器是否为一个环的代表元. 利用这一算法实现一个任意重整算法.

引理 10.15 给定了一个回置换, 有可能通过 $n + c_N - c_T$ 次赋值完成一次变动型重整, 这里的 n 是元素个数, c_N 是非平凡环的个数, 而 c_T 是平凡环的个数.

10.3 反转算法

一种简单而且有用的变动型重整是范围的*反转* (reverse). 这种重整由 n 元有穷集合上的*反转置换*导出, 集合上的反转置换基于其索引置换定义如下

$$p(i) = (n-1) - i$$

引理 10.16 在一个反转置换里, 非平凡环有 $\lfloor n/2 \rfloor$ 个; 平凡环有 $n \bmod 2$ 个.

引理 10.17 在一个置换里最多有 $\lfloor n/2 \rfloor$ 个非平凡环.

反转的概念直接由索引迭代器上的下面算法定义:[1]

```
template<typename I>
    requires(Mutable(I) && IndexedIterator(I))
void reverse_n_indexed(I f, DistanceType(I) n)
{
    // 前条件: mutable_counted_range(f, n)
    DistanceType(I) i(0);
    n = predecessor(n);
    while (i < n) {
        // n = (n_original - 1) - i
        exchange_values(f + i, f + n);
        i = successor(i);
```

1. 反转算法也可以返回其中不移动的元素的范围: 当整个范围中元素个数为奇数时返回中间元素, 元素个数为偶数时返回两个 "中间" 元素之间的空范围. 由于还没有看到需要利用这种返回值的实际情况, 因此这里返回的是 void. 当然, 对于以前向迭代器的计数范围为参数的版本, 返回其极限是很有用的.

```
        n = predecessor(n);
    }
}
```

如果把这个算法用于前向或者双向迭代器, 它将执行平方次数的迭代器增量运算. 对于双向迭代器, 每次迭代需要做两次检测:

```
template<typename I>
    requires(Mutable(I) && BidirectionalIterator(I))
void reverse_bidirectional(I f, I l)
{
    // 前条件: mutable_bounded_range(f, l)
    while (true) {
        if (f == l) return;
        l = predecessor(l);
        if (f == l) return;
        exchange_values(f, l);
        f = successor(f);
    }
}
```

如果范围的规模已知, 就可以用 reverse_swap_ranges_n:

```
template<typename I>
    requires(Mutable(I) && BidirectionalIterator(I))
void reverse_n_bidirectional(I f, I l, DistanceType(I) n)
{
    // 前条件: mutable_bounded_range(f, l) ∧ 0 ⩽ n ⩽ l − f
    reverse_swap_ranges_n(l, f, half_nonnegative(n));
}
```

reverse_swap_ranges_n 的前两个实参的顺序根据下面事实确定: 算法要在第一个范围里反向运动. 把 $n < l - f$ 送给 reverse_n_bidirectional, 使位于中间的值留在原位.

如果一种数据结构提供的是前向迭代器, 例如链接迭代器, 自然可以用 reverse_linked. 如果一些情况中有可用的额外存储, 可以采用下面算法:

```
template<typename I, typename B>
    requires(Mutable(I) && ForwardIterator(I) &&
        Mutable(B) && BidirectionalIterator(B) &&
        ValueType(I) == ValueType(B))
I reverse_n_with_buffer(I f_i, DistanceType(I) n, B f_b)
{
    // 前条件: mutable_counted_range(f_i, n)
    // 前条件: mutable_counted_range(f_b, n)
    return reverse_copy(f_b, copy_n(f_i, n, f_b).m1, f_i);
}
```

reverse_n_with_buffer 要执行 $2n$ 次赋值.

下面一些重整中用了这种方法, 先把信息拷贝一个缓存, 然后再拷贝回来. 如果没有可用的缓存, 但是在栈空间存在着对数规模的存储, 可以用一个分治法算法: 把被处理范围划分为两个部分, 先反转每个部分, 然后用 swap_ranges_n 交换这两个部分.

引理 10.18 把划分做得尽可能平均就能最大限度地减少工作量.

如果让算法返回极限值, 就可以采用一种称为*递归中辅助计算*的技术, 来优化到中点的遍历:

```
template<typename I>
    requires(Mutable(I) && ForwardIterator(I))
I reverse_n_forward(I f, DistanceType(I) n)
{
    // 前条件: mutable_counted_range(f, n)
    typedef DistanceType(I) N;
    if (n < N(2)) return f + n;
    N h = half_nonnegative(n);
    N n_mod_2 = n - twice(h);
    I m = reverse_n_forward(f, h) + n_mod_2;
```

```
    I l = reverse_n_forward(m, h);
    swap_ranges_n(f, m, h);
    return l;
}
```

reverse_n_forward 的正确性依赖于下面事实.

引理 10.19 $[0,n)$ 上的反转置换是唯一的一个满足 $i<j \Rightarrow p(j)<p(i)$ 的置换.

显然本条件对规模为 1 的范围成立. 递归调用对每个半区归纳建立这一条件. 半区之间, 以及可能跳过的中间元素之间的条件成立由 swap_ranges_n 保证.

引理 10.20 对于长度为 $n=\sum_{i=0}^{\lfloor \log n \rfloor} a_i 2^i$ 的范围, 其中 a_i 是 n 的二进制表示的第 i 个二进制位, 赋值的总次数是 $\frac{3}{2}\sum_{i=0}^{\lfloor \log n \rfloor} a_i i 2^i$.

reverse_n_forward 需要对数规模的调用栈空间. 一个具有*存储适应性* (memory-adaptive) 的算法能利用其可能获得的尽可能多的附加空间得到最好的性能, 并能通过不多的附件空间获得很大性能改进. 由这种想法得到了下面的算法, 它使用分治法, 并在子问题可以放入缓存时转到线性时间算法 reverse_n_with_buffer:

```
template<typename I, typename B>
    requires(Mutable(I) && ForwardIterator(I) &&
        Mutable(B) && BidirectionalIterator(B) &&
        ValueType(I) == ValueType(B))
I reverse_n_adaptive(I f_i, DistanceType(I) n_i,
                     B f_b, DistanceType(I) n_b)
{
    // 前条件: mutable_counted_range(f_i, n_i)
    // 前条件: mutable_counted_range(f_b, n_b)
    typedef DistanceType(I) N;
    if (n_i < N(2))
        return f_i + n_i;
    if (n_i <= n_b)
        return reverse_n_with_buffer(f_i, n_i, f_b);
```

```
    N h_i = half_nonnegative(n_i);
    N n_mod_2 = n_i - twice(h_i);
    I m_i = reverse_n_adaptive(f_i, h_i, f_b, n_b) + n_mod_2;
    I l_i = reverse_n_adaptive(m_i, h_i, f_b, n_b);
    swap_ranges_n(f_i, m_i, h_i);
    return l_i;
}
```

练习 10.6 请针对给定的范围和缓存的规模推导出一个数学公式, 描述 reverse_n_adaptive 执行赋值的总次数.

10.4 轮换算法

由索引置换 $p(i) = (i + k) \bmod n$ 定义的 n 元置换 p 称为一个 k-轮换 (rotation).

引理 10.21 一个 n 元 k-轮换的逆是一个 $(n - k)$-轮换.

索引为 i 的元素在环

$$\{i, (i + k) \bmod n, (i + 2k) \bmod n, \ldots\} = \{(i + uk) \bmod n\}$$

里. 该环的长度是满足下式的最小正整数 m

$$i = (i + mk) \bmod n$$

这等价于 $mk \bmod n = 0$, 说明环的长度与 i 无关. 由于 m 是满足 $mk \bmod n = 0$, $\operatorname{lcm}(k, n) = mk$ 的最小正整数 ($\operatorname{lcm}(a, b)$ 是 a 和 b 的最小公倍数). 利用标准等式

$$\operatorname{lcm}(a, b) \gcd(a, b) = ab$$

就可以得到环的规模

$$m = \frac{\operatorname{lcm}(k, n)}{k} = \frac{kn}{\gcd(k, n)k} = \frac{n}{\gcd(k, n)}$$

因此环的个数是 $\gcd(k, n)$.

考虑一个环中两个元素 $(i+uk) \bmod n$ 和 $(i+vk) \bmod n$. 它们的距离是

$$\begin{aligned}|(i+uk) \bmod n - (i+vk) \bmod n| &= (u-v)k \bmod n \\ &= (u-v)k - pn\end{aligned}$$

这里有 $p = \text{quotient}((u-v)k, n)$. 因为 k 和 n 都被 $d = \gcd(k,n)$ 整除, 所以该距离也被 d 整除. 由此可知, 一个环上不同元素之间的距离至少是 d, 因此索引为 $[0,d)$ 的元素属于不同的环.

范围 $[f,l)$ 中的 k-轮换重整等价于交换子范围 $[f,m)$ 和 $[m,l)$ 里相应位置的各对值, 其中 $m = f + ((l-f)-k) = l-k$. m 是比 k 更有用的输入. 如果涉及的是前向或双向迭代器, 有了它就可以避免再通过线性次操作从 k 算出 m. 返回迭代器 $m' = f+k$ 指明了位于 f 的元素的新位置, 这对许多算法也很有用.[2]

引理 10.22 围绕迭代器 m 做范围 $[f,l)$ 的轮换, 而后围绕返回值 m' 做轮换, 将会返回 m 并将该范围恢复到它原来的状态.

可以用 cycle_from 实现一个索引迭代器或随机访问迭代器范围的 k-轮换重整. 这时去置换是 $p(i) = (i+k) \bmod n$, 回置换是其逆 $p^{-1}(i) = (i+(n-k)) \bmod n$, 这里有 $n-k = m-f$. 我们希望避免计算 mod, 而且可以看到

$$p^{-1}(i) = \begin{cases} i+(n-k) & \text{if } i < k \\ i-k & \text{if } i \geqslant k \end{cases}$$

这样就得到了下面这个能用于随机访问迭代器的函数对象:

```
template<typename I>
    requires(RandomAccessIterator(I))
struct k_rotate_from_permutation_random_access
{
    DistanceType(I) k;
    DistanceType(I) n_minus_k;
    I m_prime;
```

2. Joseph Tighe 建议返回一对值 m 和 m', 以便构造出一个合法范围; 尽管这是一个有趣的建议, 它保留了所有相关信息, 但我们还没有看到这种接口的有价值应用.

```
    k_rotate_from_permutation_random_access(I f, I m, I l) :
        k(l - m), n_minus_k(m - f), m_prime(f + (l - m))
    {
        // 前条件: bounded_range(f, l) ∧ m ∈ [f, l)
    }
    I operator()(I x)
    {
        // 前条件: x ∈ [f, l)
        if (x < m_prime) return x + n_minus_k;
        else             return x - k;
    }
};
```

对索引迭代器，由于缺乏自然的序，从一个迭代器减去一个距离要多做一次或两次加法：

```
template<typename I>
    requires(IndexedIterator(I))
struct k_rotate_from_permutation_indexed
{
    DistanceType(I) k;
    DistanceType(I) n_minus_k;
    I f;
    k_rotate_from_permutation_indexed(I f, I m, I l) :
        k(l - m), n_minus_k(m - f), f(f)
    {
        // 前条件: bounded_range(f, l) ∧ m ∈ [f, l)
    }
    I operator()(I x)
    {
        // 前条件: x ∈ [f, l)
        DistanceType(I) i = x - f;
```

```
        if (i < k) return x + n_minus_k;
        else       return f + (i - k);
    }
};
```

下面过程对每个环做一次轮换:

```
template<typename I, typename F>
    requires(Mutable(I) && IndexedIterator(I) &&
        Transformation(F) && I == Domain(F))
I rotate_cycles(I f, I m, I l, F from)
{
    // 前条件: mutable_bounded_range(f, l) ∧ m ∈ [f, l]
    // 前条件: from 是 [f, l) 上的去置换
    typedef DistanceType(I) N;
    N d = gcd<N, N>(m - f, l - m);
    while (count_down(d)) cycle_from(f + d, from);
    return f + (l - m);
}
```

本算法最早发表在 Fletcher and Silver [1966], 但在上面用 cycle_from 的地方他们用的是 cycle_to. 这些过程能正确选择合适的函数对象:

```
template<typename I>
    requires(Mutable(I) && IndexedIterator(I))
I rotate_indexed_nontrivial(I f, I m, I l)
{
    // 前条件: mutable_bounded_range(f, l) ∧ f ≺ m ≺ l
    k_rotate_from_permutation_indexed<I> p(f, m, l);
    return rotate_cycles(f, m, l, p);
}

template<typename I>
    requires(Mutable(I) && RandomAccessIterator(I))
```

```
I rotate_random_access_nontrivial(I f, I m, I l)
{
    // 前条件: mutable_bounded_range(f, l) ∧ f ≺ m ≺ l
    k_rotate_from_permutation_random_access<I> p(f, m, l);
    return rotate_cycles(f, m, l, p);
}
```

这里赋值的次数是 $n + c_N - c_T = n + \gcd(n, k)$. 再说一下, n 是元素个数, c_N 是非平凡环的个数, 而 c_T 是平凡环的个数. 对于 $1 \leqslant n, k \leqslant m$, $\gcd(n, k)$ 的期望值是 $\frac{6}{\pi^2} \ln m + C + O(\frac{\ln m}{\sqrt{m}})$ (参看 Diaconis and Erdös [2004]).

下面性质引出了一个用于双向迭代器的轮换算法.

引理 10.23 $[0, n)$ 上的 k-轮换是唯一满足下面条件的置换 p: 它能逆转子范围 $[0, n-k)$ 和 $[n-k, n)$ 的相对顺序, 但维持每个子范围内部的相对顺序:

1. $i < n-k \wedge n-k \leqslant j < n \Rightarrow p(j) < p(i)$

2. $i < j < n-k \vee n-k \leqslant i < j \Rightarrow p(i) < p(j)$

反转置换满足 1 但不满足 2. 将反转应用于 $[0, n-k)$ 和 $[n-k, n)$, 然后再将反转应用于整个范围, 就能同时满足两个条件:

```
reverse_bidirectional(f, m);
reverse_bidirectional(m, l);
reverse_bidirectional(f, l);
```

可以看到, 返回值 m' 可以用 reverse_swap_ranges_bounded 得到:[3]

```
template<typename I>
    requires(Mutable(I) && BidirectionalIterator(I))
I rotate_bidirectional_nontrivial(I f, I m, I l)
{
    // 前条件: mutable_bounded_range(f, l) ∧ f ≺ m ≺ l
    reverse_bidirectional(f, m);
    reverse_bidirectional(m, l);
```

3. 用 reverse_swap_ranges_bounded 去确定 m' 是 Wilson Ho 和 Raymond Lo 的建议.

```
    pair<I, I> p = reverse_swap_ranges_bounded(m, l, f, m);
    reverse_bidirectional(p.m1, p.m0);
    if (m == p.m0) return p.m1;
    else           return p.m0;
}
```

引理 10.24 赋值次数是 $3(\lfloor n/2 \rfloor + \lfloor k/2 \rfloor + \lfloor (n-k)/2 \rfloor)$. 当 n 和 k 都是偶数时也就是 $3n$, 否则是 $3(n-1)$.

给定范围 $[f, l)$ 和该范围里的迭代器 m, 调用

$$p \leftarrow \text{swap\_ranges\_bounded}(f, m, m, l)$$

将 p 设置为一对迭代器, 使得

$$p.m0 = m \vee p.m1 = l$$

如果 $p.m0 = m \wedge p.m1 = l$, 事情就做完了. 否则 $[f, p.m0)$ 已经在最终位置, 还要根据 $p.m0 = m$ 或者 $p.m1 = l$ 决定对 $[p.m0, l)$ 分别围绕 p.m1 或者 m 做轮换. 这样立刻就得到了下面算法, 它最早出现在 Gries and Mills [1981]:

```
template<typename I>
    requires(Mutable(I) && ForwardIterator(I))
void rotate_forward_annotated(I f, I m, I l)
{
    // 前条件: mutable_bounded_range(f, l) ∧ f ≺ m ≺ l
                                        DistanceType(I) a = m - f;
                                        DistanceType(I) b = l - m;
    while (true) {
        pair<I, I> p = swap_ranges_bounded(f, m, m, l);
        if (p.m0 == m && p.m1 == l) { assert(a == b);
            return;
        }
        f = p.m0;
        if (f == m) {                   assert(b > a);
```

```
            m = p.m1;                    b = b - a;
        } else {                         assert(a > b);
                                         a = a - b;
        }
    }
}
```

引理 10.25 第一次取到 else 子句时 $f = m'$, 这是轮换的标准返回值.

标注变量 (annotation variable) a 和 b 总等于被交换的两个子范围的规模, 同时用减法求初始规模的 gcd. swap_ranges_bounded 每次执行 exchange_values 把一个值放入其最终位置; 最后一次调用 swap_ranges_bounded 的情况不同. 每调用 exchange_values 一次将两个值放入最终位置. 由于 swap_ranges_bounded 的最后调用执行 $\gcd(n, k)$ 次对 exchange_values 的调用, 对 exchange_values 总调用次数是 $n - \gcd(n, k)$.

上面引理也提出了一种实现完整的 rotate_forward 的方法: 建立代码的第二份拷贝, 在 else 子句里保存 f 的一份拷贝, 而后调用 rotate_forward_annotated 去完成轮换. 由这些想法可以得到下面两个过程:

```
template<typename I>
    requires(Mutable(I) && ForwardIterator(I))
void rotate_forward_step(I& f, I& m, I l)
{
    // 前条件: mutable_bounded_range(f, l) ∧ f ≺ m ≺ l
    I c = m;
    do {
        swap_step(f, c);
        if (f == m) m = c;
    } while (c != l);
}

template<typename I>
    requires(Mutable(I) && ForwardIterator(I))
```

```
I rotate_forward_nontrivial(I f, I m, I l)
{
    // 前条件: mutable_bounded_range(f, l) ∧ f ≺ m ≺ l
    rotate_forward_step(f, m, l);
    I m_prime = f;
    while (m != l) rotate_forward_step(f, m, l);
    return m_prime;
}
```

练习 10.7 请验证 rotate_forward_nontrivial 能围绕 m 做 [f, l) 的轮换并返回 m′.

有时做一个范围的部分轮换 (partially rotate) 也很有用, 即, 在对原位于 [f, m) 的对象做重整时把正确元素移到 [f, m′), 但维持 [m′l) 不变. 例如, 这种操作用于把一些不要的对象移到序列末端准备删除. 下面算法完成这一工作:

```
template<typename I>
    requires(Mutable(I) && ForwardIterator(I))
I rotate_partial_nontrivial(I f, I m, I l)
{
    // 前条件: mutable_bounded_range(f, l) ∧ f ≺ m ≺ l
    return swap_ranges(m, l, f);
}
```

引理 10.26 rotate_partial_nontrivial 的后条件说明它执行的是部分轮换, 对位于 [m′, l) 的对象做 k-轮换, 这里的 $k = -(l - f) \bmod (m - f)$.

rotate_partial_nontrivial 的反向版本里使用 swap_ranges 的反向版本, 该算法有时也很有用.

如果有额外的存储可用, 就可能使用下面算法:

```
template<typename I, typename B>
    requires(Mutable(I) && ForwardIterator(I) &&
        Mutable(B) && ForwardIterator(B))
I rotate_with_buffer_nontrivial(I f, I m, I l, B f_b)
{
```

```
    // 前条件: mutable_bounded_range(f, l) ∧ f ≺ m ≺ l
    // 前条件: mutable_counted_range(f_b, l − f)
    B l_b = copy(f, m, f_b);
    I m_prime = copy(m, l, f);
    copy(f_b, l_b, m_prime);
    return m_prime;
}
```

rotate_with_buffer_nontrivial 需要执行 $(l-f)+(m-f)$ 次赋值, 而下面的算法只执行 $(l-f)+(l-m)$ 次赋值. 在对一个双向迭代器范围做轮换时, 可以选择使赋值数目达到最小的算法. 当然, 在运行时计算相关的差需要执行线性次的 successor 运算:

```
template<typename I, typename B>
    requires(Mutable(I) && BidirectionalIterator(I) &&
        Mutable(B) && ForwardIterator(B))
I rotate_with_buffer_backward_nontrivial(I f, I m, I l, B f_b)
{
    // 前条件: mutable_bounded_range(f, l) ∧ f ≺ m ≺ l
    // 前条件: mutable_counted_range(f_b, l − f)
    B l_b = copy(m, l, f_b);
    copy_backward(f, m, l);
    return copy(f_b, l_b, f);
}
```

10.5 算法选择

在第 10.3 节里, 针对不同的迭代器需求和过程原型给出了几个不同的反转算法, 包括一些处理计数或有界范围的版本. 针对其他迭代器类型定义各种变形, 其中采用最方便的原型也很值得去做. 例如, 另一种常量时间的迭代器减运算提示我们写出下面的算法, 用于反转索引迭代器的有界范围:

```
template<typename I>
    requires(Mutable(I) && IndexedIterator(I))
void reverse_indexed(I f, I l)
{
    // 前条件: mutable_bounded_range(f,l)
    reverse_n_indexed(f, l - f);
}
```

如果要反转的是一个前向迭代器范围, 通常会有足够的可用存储, 使 reverse_n_adaptive 能高效工作. 如果要反转的范围规模适中, 有可能通过常规方式获得这种存储 (例如用 malloc). 当然, 如果范围的规模很大, 可能就不存在足够的物理存储器来支持足够大的缓冲区了. 由于 reverse_n_adaptive 一类算法在缓冲区与被修改的范围相比很小的情况下也能有效工作, 提供一种方式来分配临时缓冲区 (temporary buffer) 是很有意义的. 这一分配得到的存储可能小于所请求的存储. 在有虚存的系统里, 被分配内存有特定的物理内存. 这种临时内存只准备用很短时间, 而且保证在算法终止前交回.

举个例子, 下面算法用了类型为 temporary_buffer 的临时缓冲区:

```
template<typename I>
    requires(Mutable(I) && ForwardIterator(I))
void reverse_n_with_temporary_buffer(I f, DistanceType(I) n)
{
    // 前条件: mutable_counted_range(f,n)
    temporary_buffer<ValueType(I)> b(n);
    reverse_n_adaptive(f, n, begin(b), size(b));
}
```

构造函数 b(n) 分配内存, 用于保存 $m \leqslant n$ 个类型为 ValueType(I) 的相邻对象; size(b) 返回数 m, 而 begin(b) 返回指向这一范围起始位置的迭代器. b 的析构函数释放这块内存.

针对同一个问题, 经常存在能很好处理不同种类的需求的多种不同算法. 例如, 对于轮换有三种有用算法, 分别处理索引 (和随机访问) 迭代器、双向迭代器和前向迭代器. 有可能基于对类型的需求, 在一族算法中自动选择一个算

法. 我们完成这一工作时采用一种称为概念分发 (concept dispatch) 的方法. 下面从高层分发过程的定义开始, 其中也处理了一些平凡轮换:

```
template<typename I>
    requires(Mutable(I) && ForwardIterator(I))
I rotate(I f, I m, I l)
{
    // 前条件: mutable_bounded_range(f, l) ∧ m ∈ [f, l]
    if (m == f) return l;
    if (m == l) return f;
    return rotate_nontrivial(f, m, l, IteratorConcept(I)());
}
```

类型函数 IteratorConcept 返回概念标志类型 (concept tag type), 该类型编码了相应函数的参数能建模的最强概念. 在此之后就可以为每个概念标志类型实现一个过程:

```
template<typename I>
    requires(Mutable(I) && ForwardIterator(I))
I rotate_nontrivial(I f, I m, I l, forward_iterator_tag)
{
    // 前条件: mutable_bounded_range(f, l) ∧ f ≺ m ≺ l
    return rotate_forward_nontrivial(f, m, l);
}

template<typename I>
    requires(Mutable(I) && BidirectionalIterator(I))
I rotate_nontrivial(I f, I m, I l, bidirectional_iterator_tag)
{
    // 前条件: mutable_bounded_range(f, l) ∧ f ≺ m ≺ l
    return rotate_bidirectional_nontrivial(f, m, l);
}

template<typename I>
```

```
    requires(Mutable(I) && IndexedIterator(I))
I rotate_nontrivial(I f, I m, I l, indexed_iterator_tag)
{
    // 前条件: mutable_bounded_range(f,l) ∧ f ≺ m ≺ l
    return rotate_indexed_nontrivial(f, m, l);
}

template<typename I>
    requires(Mutable(I) && RandomAccessIterator(I))
I rotate_nontrivial(I f, I m, I l, random_access_iterator_tag)
{
    // 前条件: mutable_bounded_range(f,l) ∧ f ≺ m ≺ l
    return rotate_random_access_nontrivial(f, m, l);
}
```

在概念分发的定义中, 并没有考虑除概念需求之外的其他因素. 例如, 如表 10.1 中总结的, 存在着三个算法, 它们都能实现随机访问迭代器范围的轮换, 不同算法执行赋值的次数不同. 如果被轮换的范围可以放入缓存, 执行 $n + \gcd(n,k)$ 次赋值的随机访问算法将能给出最好的性能. 当处理的范围不能放入缓存时, $3n$ 次赋值的双向算法或者 $3(n - \gcd(n,k))$ 次赋值的前向算法比较快. 在这种情况下, 究竟是双向算法还是前向算法更快, 就要受到其他因素的影响. 包括双向算法里的循环结构更规范, 这一情况有可能抵消掉多做赋值的影响; 还有一些处理器体系结构的细节, 例如其缓存配置和预取逻辑. 还应该注意, 这些算法不仅执行值类型上的赋值, 还要执行迭代器操作. 如果值类型的规模更小, 其他操作的相对开销就增加了.

项目 10.1 请设计一些测试集, 针对不同的数组规模、元素规模、轮换的规模等比较各种算法的性能. 基于通过这一测试集得到的结果设计一个组合算法, 它根据有关的迭代器概念、范围的规模、轮换的规模、元素的规模、缓存的规模、能否使用临时缓冲区以及其他的相关考虑, 选择最合适的轮换算法.

项目 10.2 本章给出了两类基于位置的重整算法: 反转和轮换. 文献里还有这类算法的另外一些例子. 请为基于位置的重整开发一套分类体系, 将现有算法

表 10.1 不同轮换算法执行赋值的次数

算法	赋值
indexed, random_access	$n + \gcd(n, k)$
bidirectional	$3n$ 或 $3(n-2)$
forward	$3(n - \gcd(n, k))$
with_buffer	$n + (n - k)$
with_buffer_backward	$n + k$
partial	$3k$

注意: 这里 $n = l - f$ 而 $k = l - m$

分类, 开发出缺失的算法, 做出一个库.

10.6 总结

置换的结构使人可以设计和分析各种重整算法. 即使很简单的问题, 例如反转和轮换, 也能得到多种多样的有用算法. 合适算法的选择依赖于迭代器的需求和一些系统问题. *原地算法* (in-place algorithm) 这一理论概念很重要, 存储适应性算法提供了一类实际的替代性选择.

第 11 章

划分和归并

本章利用前几章的构件，构造一些基于谓词的和基于序的重整算法. 在展示针对前向和双向迭代器的几个划分算法后，我们将实现一个稳定划分算法. 而后引入一种二进制计数器机制，用于把自下而上的分治算法变换到迭代形式，例如可以对稳定划分做这种变换. 随后还要介绍一种稳定的具有存储适应性的归并算法，并用它构造一个高效的、具有存储适应性的、可用于前向迭代器的稳定排序算法. 前向迭代器是能完成重整的最弱概念.

11.1 划分

第 6 章介绍了基于谓词来划分一个范围的概念，以及处理这类范围的基本算法 partition_point. 现在考虑一些把任意范围转变为划分了的范围的算法.

练习 11.1 请实现一个 partitioned_at_point，它检测一个给定有界范围是否在某个特定迭代器的位置划分.

练习 11.2 请实现算法 potential_partition_point，它返回一个迭代器，指向如果真的做划分后的那个划分点.

引理 11.1 如果 $\mathrm{m} = \mathsf{potential\_partition\_point}(\mathrm{f}, \mathrm{l}, \mathrm{p})$，那么

$$\mathsf{count\_if}(\mathrm{f}, \mathrm{m}, \mathrm{p}) = \mathsf{count\_if\_not}(\mathrm{m}, \mathrm{l}, \mathrm{p})$$

换句话说，m 的两边没到位的元素个数相同.

这一引理给出了划分一个范围所需的最少赋值次数，即 $2\mathrm{n}+1$，其中 n 是 m 任意一侧未到位元素的个数：2n 次赋值用于未到位元素，另一个赋值用于临

时变量.

引理 11.2 存在 u!v! 个能划分包含 u 个假值和 v 个真值的范围的置换.

一个划分重整是*稳定的* (stable partition), 如果不满足谓词的元素的相对位置都能保持, 与此同时满足谓词的元素的相对位置也都能保持.

引理 11.3 稳定划分的结果唯一.

划分重整是*半稳定的* (semistable partition), 如果它能维持所有不满足谓词的元素的相对顺序. 下面算法实现了半稳定划分:[1]

```
template<typename I, typename P>
    requires(Mutable(I) && ForwardIterator(I) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
I partition_semistable(I f, I l, P p)
{
    // 前条件: mutable_bounded_range(f,l)
    I i = find_if(f, l, p);
    if (i == l) return i;
    I j = successor(i);
    while (true) {
        j = find_if_not(j, l, p);
        if (j == l) return i;
        swap_step(i, j);
    }
}
```

partition_semistable 的正确性由下面三个引理保证.

引理 11.4 在退出检测之前, $\mathsf{none}(f,i,p) \wedge \mathsf{all}(i,j,p)$.

引理 11.5 在退出检测之后, $p(\mathsf{source}(i)) \wedge \neg p(\mathsf{source}(j))$.

引理 11.6 在调用 swap_step 之后, $\mathsf{none}(f,i,p) \wedge \mathsf{all}(i,j,p)$.

1. Bentley [1984, 287–291 页] 将这一算法归功于 Nico Lomuto.

半稳定的根据是下面事实: 对 swap_step 的调用把一个不满足谓词的元素移到满足谓词的元素范围的前面, 这样, 不满足谓词的元素的顺序不变.

partition_semistable 只是在 swap_step 里用了一个临时对象.

令 $n = l - f$ 为范围里元素的个数, 并令 w 为跟在第一个满足谓词的元素之后的不满足谓词的元素的个数. 这样该谓词将应用 n 次, exchange_values 执行 w 次, 而迭代器做增量的次数为 $n + w$.

练习 11.3 请重写 partition_semistable, 把对 find_if_not 的调用展开并删去对 l 的多余检测.

练习 11.4 请将算法 partition_semistable 里的 swap_step(i, j) 换为 copy_step(j, i). 给出修改后的这个算法的后条件, 并为其取一个合适的名字, 并比较它的使用与 partition_semistable 的使用.

设 n 为被划分范围里元素的个数.

引理 11.7 一个返回划分点的划分重整需要应用谓词 n 次.

引理 11.8 如果一个针对非空范围的划分重整不返回划分点, 它需要应用谓词 $n - 1$ 次.[2]

练习 11.5 请实现一种对非空范围的划分重整, 它只应用谓词 $n - 1$ 次.

考虑一个范围, 其中只有一个满足谓词的元素, 它之后是 n 个不满足谓词的元素. partition_semistable 将执行 n 次 exchange_values 调用, 而实际上做一次就够了. 如果组合使用对满足谓词元素的前向搜索和对不满足谓词元素的后向搜索, 就可以避免不必要的交换. 下面这个算法需要双向迭代器:

```
template<typename I, typename P>
    requires(Mutable(I) && BidirectionalIterator(I) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
I partition_bidirectional(I f, I l, P p)
{
    // 前条件: mutable_bounded_range(f, l)
    while (true) {
```

2. 这一引理和紧随其后的练习来自 Jon Brandt 给我们的建议.

```
        f = find_if(f, l, p);
        l = find_backward_if_not(f, l, p);
        if (f == l) return f;
        reverse_swap_step(l, f);
    }
}
```

与 partition_semistable 一样, partition_bidirectional 也只用了一个临时对象.

引理 11.9 执行 exchange_values 的次数 ν, 等于位置不对的不满足谓词的元素的个数. 因此赋值的总次数是 3ν.

练习 11.6 请为前向迭代器实现一个划分重整算法, 在算出了划分点之后, 它调用 exchange_values 的次数与 partition_bidirectional 一样.

有可能用另一个不同的重整完成一种划分, 其中只做一遍循环, 而且只用 $2\nu+1$ 次赋值. 其想法是保存起第一个错位元素, 留下一个 "空位", 而后反复交替地在划分点的两边找到的错位元素, 将其移到空位并从而做出一个新的空位, 最终将保存的元素移入最后的空位.

练习 11.7 请采用上述技术实现 partition_single_cycle.

练习 11.8 请为双向迭代器实现一种划分重整, 它找出适当的卫兵 (sentinel) 元素, 而后用 find_if_unguarded 和 find_backward_if_not 的不带条件检查的版本.

练习 11.9 重做上面练习, 把一遍循环的技术结合进来.

双向划分算法及一遍循环的带卫兵划分算法, 都来自 C. A .R. Hoare.[3]

如果划分的两边都需要稳定性, 而且有与原范围一样大的内存可用, 就可以采用下面的算法:

```
template<typename I, typename B, typename P>
    requires(Mutable(I) && ForwardIterator(I) &&
        Mutable(B) && ForwardIterator(B) &&
```

3. 见 Hoare [1962] 有关 Quicksort 算法的论述. 由于 Quicksort 的需要, Hoare 的划分算法交换的是大于或等于给定元素的元素与小于或等于给定元素的元素. 相等元素的范围将从中间分开. 请注意, 关系 $\leqslant$ 和 $\geqslant$ 并不互补.

```
        ValueType(I) == ValueType(B) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
I partition_stable_with_buffer(I f, I l, B f_b, P p)
{
    // 前条件: mutable_bounded_range(f, l)
    // 前条件: mutable_counted_range(f_b, l - f)
    pair<I, B> x = partition_copy(f, l, f, f_b, p);
    copy(f_b, x.m1, x.m0);
    return x.m0;
}
```

如果没有足够的内存用于完整大小的缓冲区, 可以采用一种分治算法实现稳定划分. 如果范围里只有一个元素, 它自然是排好序的, 其划分点可以通过应用一次谓词得到:

```
template<typename I, typename P>
    requires(Mutable(I) && ForwardIterator(I) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
pair<I, I> partition_stable_singleton(I f, P p)
{
    // 前条件: readable_bounded_range(f, successor(f))
    I l = successor(f);
    if (!p(source(f))) f = l;
    return pair<I, I>(f, l);
}
```

返回的值是划分点和范围的极限. 换句话说, 这里返回的是满足谓词的值的范围.

两个邻接的划分范围可以组合为一个更大的划分范围, 方法是将由两个划分点确定的范围围绕着该范围的中点旋转:

```
template<typename I>
    requires(Mutable(I) && ForwardIterator(I))
pair<I, I> combine_ranges(const pair<I, I>& x,
```

```
                                const pair<I, I>& y)
{
    // 前条件: mutable_bounded_range(x.m0, y.m0)
    // 前条件: x.m1 ∈ [x.m0, y.m0]
    return pair<I, I>(rotate(x.m0, x.m1, y.m0), y.m1);
}
```

引理 11.10 combine_ranges 对三个互不重叠的范围的应用是可结合的.

引理 11.11 如果对某个谓词 p 有,

$$(\forall i \in [x.m0, x.m1))\, p(i) \wedge$$
$$(\forall i \in [x.m1, y.m0))\, \neg p(i) \wedge$$
$$(\forall i \in [y.m0, y.m1))\, p(i)$$

那么在

$$z \leftarrow \text{combine\_ranges}(x, y)$$

之后下面条件成立:

$$(\forall i \in [x.m0, z.m0))\, \neg p(i)$$
$$(\forall i \in [z.m0, z.m1))\, p(i)$$

如果输入是满足谓词的值的范围, 输出也是. 这样, 一个非单个元素的范围可以通过如下方法完成稳定划分: 将它从中间分割为两半, 递归地划分这两个半长范围, 而后将划分后的部分组合到一起:

```
template<typename I, typename P>
    requires(Mutable(I) && ForwardIterator(I) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
pair<I, I> partition_stable_n_nonempty(I f, DistanceType(I) n, P p)
{
    // 前条件: mutable_counted_range(f, n) ∧ n > 0
    if (one(n)) return partition_stable_singleton(f, p);
    DistanceType(I) h = half_nonnegative(n);
```

```
    pair<I, I> x = partition_stable_n_nonempty(f, h, p);
    pair<I, I> y = partition_stable_n_nonempty(x.m1, n - h, p);
    return combine_ranges(x, y);
}
```

由于对任何多于一个元素的范围的二分分割都不会出现空范围, 因此只需要在最上层处理空范围的情况:

```
template<typename I, typename P>
    requires(Mutable(I) && ForwardIterator(I) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
pair<I, I> partition_stable_n(I f, DistanceType(I) n, P p)
{
    // 前条件: mutable_counted_range(f, n)
    if (zero(n)) return pair<I, I>(f, f);
    return partition_stable_n_nonempty(f, n, p);
}
```

在递归的最下层需要执行正好 n 次谓词应用. partition_stable_n_nonempty 递归的深度是 $\lceil \log_2 n \rceil$. 在每个递归层面上, 平均需要旋转 $n/2$ 个元素, 需要做 $n/2$ 到 $3n/2$ 次赋值 (依赖于迭代器的类别). 对随机访问迭代器, 总赋值次数是 $n \log_2 n/2$, 而对前向和双向迭代器总赋值次数是 $3n \log_2 n/2$.

练习 11.10 请利用前一章的技术, 做出 partition_stable_n 的一个具有存储适应性的版本.

11.2 平衡的归约

虽然 partition_stable_n 的性能依赖于所处理的范围里的中点分割, 但其正确性并不依赖于这一点. 因为 combine_ranges 是一个部分可结合运算, 因此其子段分割可以在任何位置进行. 利用这一事实可以做出一个性能类似的迭代算法. 在一些情况下, 例如事先不知道范围的规模, 或者希望消除过程调用的开销, 这样的算法非常有用. 这里的基本想法是进行归约, 将 partition_stable_singleton 应用于每个单个元素的范围, 而后用 combine_ranges 组合得到的结果:

```
reduce_nonempty(
    f, l,
    combine_ranges<I>,
    partition_trivial<I, P>(p));
```

partition_trivial 是一个函数对象, 把谓词的参数绑定到 partition_stable_singleton:

```
template<typename I, typename P>
    requires(ForwardIterator(I) &&
    UnaryPredicate(P) && ValueType(I) == Domain(P))
struct partition_trivial
{
    P p;
    partition_trivial(const P & p) : p(p) { }
    pair<I, I> operator()(I i)
    {
        return partition_stable_singleton<I, P>(i, p);
    }
};
```

如果在这里调用 reduce_nonempty, 就会导致平方复杂性. 现在要利用运算的部分可结合性, 建立起一棵平衡的归约树. 下面将使用二进制计数器技术, 自下而上地构造这棵归约树.[4] 对硬件二进制计数器的增量操作给一个 n-位的二进制整数加 1. 在位置 i 的 1 具有 2^i 的权重 (weight); 从这个位置的进位具有权值 2^{i+1} 并传到下一高位. 这里要用计数器中位置 i 的 "位" 表示或者是空, 或者是从原范围归约 2^i 个元素的结果. 当进位传到下一高位时, 它或者被存下, 或者与另一具有同样权的值组合. 从最高位的进位被下面过程返回, 该过程以单位元作为一个显式参数, 与 reduce_nonzeroes 一样:

```
template<typename I, typename Op>
    requires(Mutable(I) && ForwardIterator(I) &&
        BinaryOperation(Op) && ValueType(I) == Domain(Op))
```

4. Knuth [1998, 5.2.4 节 (归并排序), 练习 17, 167 页] 里把这一技术归功于 John McCarthy.

```
Domain(Op) add_to_counter(I f, I l, Op op, Domain(Op) x,
                          const Domain(Op)& z)
{
    if (x == z) return z;
    while (f != l) {
        if (source(f) == z) {
            sink(f) = x;
            return z;
        }
        x = op(source(f), x);
        sink(f) = z;
        f = successor(f);
    }
    return x;
}
```

计数器使用的存储由下面类型提供, 该类型还处理来自 add_to_counter 的溢出, 方式就是扩展计数器:

```
template<typename Op>
    requires(BinaryOperation(Op))
struct counter_machine
{
    typedef Domain(Op) T;
    Op op;
    T z;
    T f[64];
    pointer(T) l;
    counter_machine(Op op, const Domain(Op)& z) :
        op(op), z(z), l(f) { }
    void operator()(const T& x)
    {
```

```
        // 前条件: 不能被调用多于 2^64 − 1 次
        T tmp = add_to_counter(f, l, op, x, z);
        if (tmp != z) {
            sink(l) = tmp;
            l = successor(l);
        }
    }
};
```

这里用了一个 C++ 的数组; 也可以考虑其他实现方式.[5]

在对范围里的每个元素调用 add_to_counter 之后, 用最左归约的方式组合起计数器里的所有非空位置, 就能得到最后结果:

```
template<typename I, typename Op, typename F>
    requires(Iterator(I) && BinaryOperation(Op) &&
        UnaryFunction(F) && I == Domain(F) &&
        Codomain(F) == Domain(Op))
Domain(Op) reduce_balanced(I f, I l, Op op, F fun,
                           const Domain(Op)& z)
{
    // 前条件: bounded_range(f, l) ∧ l − f < 2^64
    // 前条件: partially_associative(op)
    // 前条件: (∀x ∈ [f, l)) fun(x) 有定义
    counter_machine<Op> c(op, z);
    while (f != l) {
        c(fun(f));
        f = successor(f);
    }
    transpose_operation<Op> t_op(op);
    return reduce_nonzeroes(c.f, c.l, t_op, z);
}
```

5. 选用 64 个元素的数组可以处理 64-位体系结构上的任何应用程序.

计数器里更高位的那些值对应于原范围的较早元素,而且这里的运算不是可交换的. 因此必须采用该运算的一个反转的版本,该版本可以通过下面函数对象得到:

```
template<typename Op>
    requires(BinaryOperation(Op))
struct transpose_operation
{
    Op op;
    transpose_operation(Op op) : op(op) { }
    typedef Domain(Op) T;
    T operator()(const T& x, const T& y)
    {
        return op(y, x);
    }
};
```

现在就可以用迭代的方式实现稳定划分了,下面是这个过程:

```
template<typename I, typename P>
    requires(ForwardIterator(I) && UnaryPredicate(P) &&
        ValueType(I) == Domain(P))
I partition_stable_iterative(I f, I l, P p)
{
    // 前条件: bounded_range(f, l) ∧ l − f < 2^64
    return reduce_balanced(
        f, l,
        combine_ranges<I>,
        partition_trivial<I, P>(p),
        pair<I, I>(f, f)
      ).m0;
}
```

$\text{pair}_{I,I}(f,f)$ 是表示单位元的好办法, 因为它绝不会被 partition_trivial 或上面的组合运算返回.

这个迭代算法构造出的归约树与前面的递归过程不同. 当问题的大小正好等于 2^k 时, 递归的和迭代的算法版本将执行同样的组合序列; 否则, 迭代的版本可能多做至多线性数量的额外工作. 例如, 在某些情况下, 算法的复杂性会从 $n\log_2 n$ 变为 $n\log_2 n + \frac{n}{2}$.

练习 11.11 利用 reduce_balanced 实现第 8 章 sort_linked_nonempty_n 的迭代版本.

练习 11.12 利用 reduce_balanced 实现第 10 章 reverse_n_adaptive 的迭代版本.

练习 11.13 利用 reduce_balanced 实现 partition_stable_n 的迭代的和具有存储适应性的版本.

11.3 归并

第 9 章给出了一些拷贝式的归并算法, 它们将两个上升范围归并到第三个上升范围. 对排序而言, 将两个毗邻的上升范围归并为一个上升范围的重整是很有用的操作. 如果有一块与第一个范围同样大的缓冲区, 就可以用下面过程:[6]

```
template<typename I, typename B, typename R>
    requires(Mutable(I) && ForwardIterator(I) &&
        Mutable(B) && ForwardIterator(B) &&
        ValueType(I) == ValueType(B) &&
        Relation(R) && ValueType(I) == Domain(R))
I merge_n_with_buffer(I f0, DistanceType(I) n0,
                      I f1, DistanceType(I) n1, B f_b, R r)
{
    // 前条件: mergeable(f0, n0, f1, n1, r)
    // 前条件: mutable_counted_range(f_b, n0)
    copy_n(f0, n0, f_b);
    return merge_copy_n(f_b, n0, f1, n1, f0, r).m2;
```

6. 练习 9.5 的解可以解释为什么需要提取成员 m2.

}

其中 mergeable 的定义如下:

$$\begin{aligned}
&\textbf{property}(\mathrm{I} : \mathit{ForwardIterator}, \mathrm{N} : \mathit{Integer}, \mathrm{R} : \mathit{Relation})\\
&\quad \textbf{requires}(\mathit{Mutable}(\mathrm{I}) \wedge \mathsf{ValueType}(\mathrm{I}) = \mathsf{Domain}(\mathrm{R}))\\
&\mathsf{mergeable} : \mathrm{I} \times \mathrm{N} \times \mathrm{I} \times \mathrm{N} \times \mathrm{R}\\
&\quad (\mathrm{f}_0, \mathrm{n}_0, \mathrm{f}_1, \mathrm{n}_1, \mathrm{r}) \mapsto \mathrm{f}_0 + \mathrm{n}_0 = \mathrm{f}_1 \wedge\\
&\qquad\qquad \mathsf{mutable_counted_range}(\mathrm{f}_0, \mathrm{n}_0 + \mathrm{n}_1) \wedge\\
&\qquad\qquad \mathsf{weak_ordering}(\mathrm{r}) \wedge\\
&\qquad\qquad \mathsf{increasing_counted_range}(\mathrm{f}_0, \mathrm{n}_0, \mathrm{r}) \wedge\\
&\qquad\qquad \mathsf{increasing_counted_range}(\mathrm{f}_1, \mathrm{n}_1, \mathrm{r})
\end{aligned}$$

引理 11.12 merge_n_with_buffer 的后条件是

$$\mathsf{increasing\_counted\_range}(\mathrm{f}_0, \mathrm{n}_0 + \mathrm{n}_1, \mathrm{r})$$

一个归并是稳定的, 如果其输出范围里维持了各输入范围里的所有相等元素的相对顺序, 也维持了第一和第二个范围里相等的元素的相对顺序.

引理 11.13 merge_n_with_buffer 是稳定的.

注意, merge_linked_nonempty、merge_copy 和 merge_copy_backward 也都稳定.

可以用一半规模的缓冲区完成对一个范围的排序:[7]

```
template<typename I, typename B, typename R>
    requires(Mutable(I) && ForwardIterator(I) &&
        Mutable(B) && ForwardIterator(B) &&
        ValueType(I) == ValueType(B) &&
        Relation(R) && ValueType(I) == Domain(R))
I sort_n_with_buffer(I f, DistanceType(I) n, B f_b, R r)
{
    // 前条件: mutable_counted_range(f, n) ∧ weak_ordering(r)
    // 前条件: mutable_counted_range(f_b, ⌈n/2⌉)
```

7. John W. Mauchly 在其报告 "Sorting and collating" [Mauchly 1946]A 里第一次描述了一个类似算法.

```
    DistanceType(I) h = half_nonnegative(n);
    if (zero(h)) return f + n;
    I m = sort_n_with_buffer(f, h,     f_b, r);
          sort_n_with_buffer(m, n - h, f_b, r);
    return merge_n_with_buffer(f, h, m, n - h, f_b, r);
}
```

引理 11.14 sort_n_with_buffer 的后条件是

$$\text{increasing\_counted\_range}(f, n, r)$$

一个排序算法是稳定的, 如果它能维持等值元素的相对顺序.

引理 11.15 sort_n_with_buffer 是稳定的.

这一算法递归 $\lceil \log_2 n \rceil$ 层. 每层执行至多 $3n/2$ 个赋值, 总数受囿于 $\frac{3}{2} n \lceil \log_2 n \rceil$. 在自下而上数的第 i 层, 最坏情况下的比较次数是 $n - \frac{n}{2^i}$, 这就给出了比较次数的上界:

$$n\lceil \log_2 n \rceil - \sum_{i=1}^{\lceil \log_2 n \rceil} \frac{n}{2^i} \approx n\lceil \log_2 n \rceil - n$$

如果有足够大的缓冲区可用, sort_n_with_buffer 是一个高效算法. 如果可用的存储较少, 可以用一个具有存储适应性的算法. 把第一个子范围从中间割开, 再用这一中间元素作为下界分割第二个子范围, 结果将得到四个子范围 r_0、r_1、r_2 和 r_3, 其中 r_2 里的值严格小于 r_1 里的值. 旋转 r_1 和 r_2 就得到了两个归并子问题: (r_0 和 r_2 归并, r_1 和 r3 归并):

```
template<typename I, typename R>
    requires(Mutable(I) && ForwardIterator(I) &&
        Relation(R) && ValueType(I) == Domain(R))
void merge_n_step_0(I f0, DistanceType(I) n0,
                    I f1, DistanceType(I) n1, R r,
                    I& f0_0, DistanceType(I)& n0_0,
                    I& f0_1, DistanceType(I)& n0_1,
                    I& f1_0, DistanceType(I)& n1_0,
```

```
                    I& f1_1, DistanceType(I)& n1_1)
{
    // 前条件: mergeable(f_0, n_0, f_1, n_1, r)
    f0_0 = f0;
    n0_0 = half_nonnegative(n0);
    f0_1 = f0_0 + n0_0;
    f1_1 = lower_bound_n(f1, n1, source(f0_1), r);
    f1_0 = rotate(f0_1, f1, f1_1);
    n0_1 = f1_0 - f0_1;
    f1_0 = successor(f1_0);
    n1_0 = predecessor(n0 - n0_0);
    n1_1 = n1 - n0_1;
}
```

引理 11.16 这里的轮换并不改变等值元素的相对位置.

在一个范围里, 迭代器 i 是一个**枢轴** (pivot), 如果它的值不小于它前面的任何值, 而且不大于它之后的任何值.

引理 11.17 在 merge_n_step_0 之后 f1_0 是一个枢轴.

也可以用 upper_bound 从右边开始做类似分割:

```
template<typename I, typename R>
    requires(Mutable(I) && ForwardIterator(I) &&
        Relation(R) && ValueType(I) == Domain(R))
void merge_n_step_1(I f0, DistanceType(I) n0,
                    I f1, DistanceType(I) n1, R r,
                    I& f0_0, DistanceType(I)& n0_0,
                    I& f0_1, DistanceType(I)& n0_1,
                    I& f1_0, DistanceType(I)& n1_0,
                    I& f1_1, DistanceType(I)& n1_1)
{
    // 前条件: mergeable(f_0, n_0, f_1, n_1, r)
```

```
    f0_0 = f0;
    n0_1 = half_nonnegative(n1);
    f1_1 = f1 + n0_1;
    f0_1 = upper_bound_n(f0, n0, source(f1_1), r);
    f1_1 = successor(f1_1);
    f1_0 = rotate(f0_1, f1, f1_1);
    n0_0 = f0_1 - f0_0;
    n1_0 = n0 - n0_0;
    n1_1 = predecessor(n1 - n0_1);
}
```

这样就得到了下面的算法 (来自 Dudziński and Dydek [1981]):

```
template<typename I, typename B, typename R>
    requires(Mutable(I) && ForwardIterator(I) &&
        Mutable(B) && ForwardIterator(B) &&
        ValueType(I) == ValueType(B) &&
        Relation(R) && ValueType(I) == Domain(R))
I merge_n_adaptive(I f0, DistanceType(I) n0,
                   I f1, DistanceType(I) n1,
                   B f_b, DistanceType(B) n_b, R r)
{
    // 前条件: mergeable(f0, n0, f1, n1, r)
    // 前条件: mutable_counted_range(f_b, n_b)
    typedef DistanceType(I) N;
    if (zero(n0) || zero(n1)) return f0 + n0 + n1;
    if (n0 <= N(n_b))
        return merge_n_with_buffer(f0, n0, f1, n1, f_b, r);
    I f0_0; I f0_1; I f1_0; I f1_1;
    N n0_0; N n0_1; N n1_0; N n1_1;
    if (n0 < n1) merge_n_step_0(
                            f0, n0, f1, n1, r,
```

```
                        f0_0, n0_0, f0_1, n0_1,
                        f1_0, n1_0, f1_1, n1_1);
    else          merge_n_step_1(
                        f0, n0, f1, n1, r,
                        f0_0, n0_0, f0_1, n0_1,
                        f1_0, n1_0, f1_1, n1_1);
          merge_n_adaptive(f0_0, n0_0, f0_1, n0_1,
                        f_b, n_b, r);
    return merge_n_adaptive(f1_0, n1_0, f1_1, n1_1,
                        f_b, n_b, r);
}
```

引理 11.18 merge_n_adaptive 终止时将得到一个递增范围.

引理 11.19 merge_n_adaptive 是稳定的.

引理 11.20 至多有 $\lfloor \log_2(\min(n0, n1)) \rfloor + 1$ 层递归.

利用 merge_n_adaptive 可以实现下面排序算法:

```
template<typename I, typename B, typename R>
    requires(Mutable(I) && ForwardIterator(I) &&
        Mutable(B) && ForwardIterator(B) &&
        ValueType(I) == ValueType(B) &&
        Relation(R) && ValueType(I) == Domain(R))
I sort_n_adaptive(I f, DistanceType(I) n,
                  B f_b, DistanceType(B) n_b, R r)
{
    // 前条件: mutable_counted_range(f, n) ∧ weak_ordering(r)
    // 前条件: mutable_counted_range(f_b, n_b)
    DistanceType(I) h = half_nonnegative(n);
    if (zero(h)) return f + n;
    I m = sort_n_adaptive(f, h,     f_b, n_b, r);
          sort_n_adaptive(m, n - h, f_b, n_b, r);
```

```
    return merge_n_adaptive(f, h, m, n - h, f_b, n_b, r);
}
```

练习 11.14 请确定一个公式, 它将赋值次数和比较次数描述为输入和缓冲区的规模的函数. 文献 Dudziński and Dydek [1981] 包含了对无缓冲区时的复杂性的细致分析.

我们可以总结出下面算法:

```
template<typename I, typename R>
    requires(Mutable(I) && ForwardIterator(I) &&
        Relation(R) && ValueType(I) == Domain(R))
I sort_n(I f, DistanceType(I) n, R r)
{
    // 前条件: mutable_counted_range(f, n) ∧ weak_ordering(r)
    temporary_buffer<ValueType(I)> b(half_nonnegative(n));
    return sort_n_adaptive(f, n, begin(b), size(b), r);
}
```

它只有最低的迭代器要求, 是稳定的, 即使 temporary_buffer 只能分配到很小百分比的所请求存储, 它也是高效的.

项目 11.1 请基于各种抽象组件开发一个排序算法库, 设计一些标准测试, 根据不同的数组大小、元素大小和缓冲区大小, 分析这些算法的性能. 给这个库写出文档, 包括各算法适合在什么环境中使用的建议.

11.4 总结

复杂的算法可以分解为一些较简单的抽象组件, 它们带有细致定义的接口. 这样发现的组件后来可以用于实现其他算法. 这种从复杂到简单, 然后再转回去的迭代过程, 是发现有效组件的系统化分类体系的核心工作.

第 12 章

复合对象

第 6 到第 11 章展示了一批算法，它们都通过迭代器或坐标结构作用于对象的汇集 (数据结构)，其行为与这些对象汇集的构造、析构和结构变动无关，因为算法中并没有把对象汇集本身看作对象. 本章将给出一些复合对象的例子，从二元组和常量规模的数组开始，直至动态序列的一系列实现. 这里要描述一种复合对象包含其他对象作为其组成部分的通用模式. 最后要展示一种机制，它使我们可以有效地描述嵌套的复合对象上的重整算法.

12.1 简单复合对象

为了理解如何把规范性延拓到复合对象，这里的讨论将从一些简单情况开始. 第 1 章介绍了类型构造子 pair，给它一对类型 T_0 和 T_1，它返回一个结构类型 pair_{T_0,T_1}. 现在用一个结构模板和若干全局过程实现 pair:

```
template<typename T0, typename T1>
    requires(Regular(T0) && Regular(T1))
struct pair
{
    T0 m0;
    T1 m1;
    pair() { } //默认构造函数
    pair(const T0& m0, const T1& m1) : m0(m0), m1(m1) { }
};
```

C++ 保证默认构造函数 (default constructor) 执行两个成员的构造工作, 保证它们能达到部分完成的状态, 而后可以赋值和销毁. C++ 自动生成拷贝构造函数 (copy constructor) 和赋值, 它们都逐个拷贝或赋值对象的各个成员; 还自动生成一个析构函数 (destructor), 它将调用各成员的析构函数. 相等和序判断则需要手工给出:

```
template<typename T0, typename T1>
    requires(Regular(T0) && Regular(T1))
bool operator==(const pair<T0, T1>& x, const pair<T0, T1>& y)
{
    return x.m0 == y.m0 && x.m1 == y.m1;
}

template<typename T0, typename T1>
    requires(TotallyOrdered(T0) && TotallyOrdered(T1))
bool operator<(const pair<T0, T1>& x, const pair<T0, T1>& y)
{
    return x.m0 < y.m0 || (!(y.m0 < x.m0) && x.m1 < y.m1);
}
```

练习 12.1 请为 $\mathsf{pair}_{T0,T1}$ 实现一种默认的序关系, less, 其中利用 T0 和 T1 的默认序, 用于两个成员类型都不是全序集的情况.

练习 12.2 请实现 $\mathsf{triple}_{T_0,T_1,T_2}$.

pair 是一种异类型 (heterogeneous type) 的类型构造子, 而 array_k 是一种同类型的 (homogeneous type) 构造子, 给定整数 k 和类型 T, 它返回常量规模的序列类型 $\mathsf{array\_k}_{k,T}$:

```
template<int k, typename T>
    requires(0 < k && k <= MaximumValue(int) / sizeof(T) &&
        Regular(T))
struct array_k
{
    T a[k];
```

```
    T& operator[](int i)
    {
        // 前条件: 0 ⩽ i < k
        return a[i];
    }
};
```

对 k 的要求基于类型的属性描述. MaximumValue(N) 返回整数类型 N 能表示的最大值, sizeof 是内在的类型属性, 返回类型的规模. C++ 为 array_k 生成语义正确的默认构造函数、拷贝构造函数、赋值和析构函数. 用于读写 x[i] 的成员函数需要自己实现.[1]

IteratorType(array_$k_{k,T}$) 定义为指向 T 的指针. 这里提供两个过程, 分别返回数组里的第一个元素和元素的极限:[2]

```
template<int k, typename T>
    requires(Regular(T))
pointer(T) begin(array_k<k, T>& x)
{
    return addressof(x.a[0]);
}

template<int k, typename T>
    requires(Regular(T))
pointer(T) end(array_k<k, T>& x)
{
    return addressof(x.a[k]);
}
```

array_$k_{k,T}$ 类型的对象 x 可以用下面代码初始化为计数范围 $[\![f, k)$ 的拷贝

```
copy_n(f, k, begin(x));
```

---

1. 就像 begin 和 end 一样, 重载的常数也需要完整的实现.

2. 完整的实现还需提供常量迭代器类型, 将其定义为指向 T 的常量指针, 还需定义在 array_k 常量上重载的 begin 和 end, 它们的返回类型是常量迭代器.

我们还不知道如何实现一种合适的能完成初始化的构造函数, 使之可以不自动构造数组里的每个元素. 另外, 前面定义的 copy_n 可以接受各种迭代器并返回极限迭代器. 但是我们没办法让拷贝构造函数返回极限迭代器.

数组上的相等和序可以用第 7 章的字典序的扩展:

```
template<int k, typename T>
    requires(Regular(T))
bool operator==(const array_k<k, T>& x, const array_k<k, T>& y)
{
    return lexicographical_equal(begin(x), end(x),
                                 begin(y), end(y));
}

template<int k, typename T>
    requires(Regular(T))
bool operator<(const array_k<k, T>& x, const array_k<k, T>& y)
{
    return lexicographical_less(begin(x), end(x),
                                begin(y), end(y));
}
```

练习 12.3 请为 $\text{array\_k}_{k,T}$ 实现 $=$ 和 $<$, 使之能为很小的 k 生成 inline 展开的代码.

练习 12.4 请为 $\text{array\_k}_{k,T}$ 实现默认序 less.

下面过程返回数组元素的个数:

```
template<int k, typename T>
    requires(Regular(T))
int size(const array_k<k, T>& x)
{
    return k;
}
```

另一过程判断其规模是否为 0:

```
template<int k, typename T>
    requires(Regular(T))
bool empty(const array_k<k, T>& x)
{
    return false;
}
```

还需不厌其烦地去为 array_k 定义 size 和 empty, 使之可以建模 *Sequence*. 这些事待会再做.

练习 12.5 请扩充 array_k, 使之能接受 $k = 0$ 的情况.

array_k 建模概念 *Linearizable* (可线性化):

$$
\begin{aligned}
&\mathit{Linearizable}(W) \triangleq \\
&\quad \mathit{Regular}(W) \\
&\quad \wedge\ \mathsf{IteratorType} : \mathit{Linearizable} \to \mathit{Iterator} \\
&\quad \wedge\ \mathsf{ValueType} : \mathit{Linearizable} \to \mathit{Regular} \\
&\qquad\quad W \mapsto \mathsf{ValueType}(\mathsf{IteratorType}(W)) \\
&\quad \wedge\ \mathsf{SizeType} : \mathit{Linearizable} \to \mathit{Integer} \\
&\qquad\quad W \mapsto \mathsf{DistanceType}(\mathsf{IteratorType}(W)) \\
&\quad \wedge\ \mathsf{begin} : W \to \mathsf{IteratorType}(W) \\
&\quad \wedge\ \mathsf{end} : W \to \mathsf{IteratorType}(W) \\
&\quad \wedge\ \mathsf{size} : W \to \mathsf{SizeType}(W) \\
&\qquad\quad \mathrm{x} \mapsto \mathsf{end}(\mathrm{x}) - \mathsf{begin}(\mathrm{x}) \\
&\quad \wedge\ \mathsf{empty} : W \to \mathsf{bool} \\
&\qquad\quad \mathrm{x} \mapsto \mathsf{begin}(\mathrm{x}) = \mathsf{end}(\mathrm{x}) \\
&\quad \wedge\ [\,] : W \times \mathsf{SizeType}(W) \to \mathsf{ValueType}(W)\& \\
&\qquad\quad (w, \mathrm{i}) \mapsto \mathsf{deref}(\mathsf{begin}(w) + \mathrm{i})
\end{aligned}
$$

empty 判断只需常量时间, 即使对那些 size 需要线性时间的情况. $w[\mathrm{i}]$ 的前条件是 $0 \leqslant \mathrm{i} < \mathsf{size}(w)$; 其复杂性由精化 *Linearizable* 概念规范的迭代器类型确定: 对前向和双向迭代器是线性时间, 对指标和随机访问迭代器是常量时间.

可线性化的类型用一对标准函数 begin 和 end 描述一个迭代器范围, 但与 array_k 的情况不同, 拷贝一个线性化的实体不必拷贝其基础元素. 正如在下面将要看到的, 这种实体不是容器 (container), 不是拥有一批元素的序列. 举例说, 下面类型建模 *Linearizable*, 但它不是容器, 而只是指定了位于某个数据结构里的迭代器的一个有界范围:

```
template<typename I>
    requires(Readable(I) && Iterator(I))
struct bounded_range {
    I f;
    I l;
    bounded_range() { }
    bounded_range(const I& f, const I& l) : f(f), l(l) { }
    const ValueType(I)& operator[](int i)
    {
        // 前条件: 0 ≤ i < l − f
        return source(f + i);
    }
};
```

C++ 自动为其生成拷贝构造函数、赋值和析构函数, 它们的语义与 $\text{pair}_{I,J}$ 的情况一样. 如果 T 是 bounded_range_I, 那么 IteratorType(T) 就定义为 I, 而 SizeType(T) 定义为 DistanceType(I).

与迭代器有关的过程可以直截了当地定义:

```
template<typename I>
    requires(Readable(I) && Iterator(I))
I begin(const bounded_range<I>& x) { return x.f; }

template<typename I>
    requires(Readable(I) && Iterator(I))
I end(const bounded_range<I>& x) { return x.l; }

template<typename I>
```

```
    requires(Readable(I) && Iterator(I))
DistanceType(I) size(const bounded_range<I>& x)
{
    return end(x) - begin(x);
}

template<typename I>
    requires(Readable(I) && Iterator(I))
bool empty(const bounded_range<I>& x)
{
    return begin(x) == end(x);
}
```

与 array_k 不同, bounded_range 的相等判断不采用字典序相等, 而是把对象作为一对迭代器来处理, 直接比较对应的值:

```
template<typename I>
    requires(Readable(I) && Iterator(I))

bool operator==(const bounded_range<I>& x,
                const bounded_range<I>& y)
{
    return begin(x) == begin(y) && end(x) == end(y);
}
```

这样定义的相等与 C++ 生成的拷贝构造函数一致, C++ 也将它处理为一对迭代器. 考虑一个建模 *Linearizable* 的类型 W. 如果 W 是一个具有线性坐标结构的容器, lexicographical_equal 是它正确的相等判断, 就像前面针对 array_k 的定义那样. 如果 W 是一个同类容器, 其坐标结构不是线性的 (例如一棵树或一个矩阵), 那么 lexicographical_equal 或者范围相等 (如前面为 bounded_range 定义的判断) 都不是正确的相等判断, 虽然 lexicographical_equal 可能仍然是有用的算法. 如果 W 不是容器, 而只不过是另一数据结构所拥有的一个范围的描述, 范围相等就是正确的相等判断了.

对于 bounded_range$_I$ 的默认全序是在一对迭代器上定义的字典序, 采用 I 上的默认全序:

```
template<typename I>
    requires(Readable(I) && Iterator(I))
struct less< bounded_range<I> >
{
    bool operator()(const bounded_range<I>& x,
                    const bounded_range<I>& y)
    {
        less<I> less_I;
        return less_I(begin(x), begin(y)) ||
               (!less_I(begin(y), begin(x)) &&
                 less_I(end(x), end(y)));
    }
};
```

即使一个迭代器类型没有自然的全序, 也要为它提供一个全序: 例如, 把迭代器的二进制表示看作一个无符号整数.

pair 和 array_k 都是广义的*复合对象* (composite object) 的具体实例. 说一个对象是*复合对象*, 如果它是由其他对象构造起来的, 那些对象称为它的*组分* (part). 整体–组分关系满足四条性质: *联系性*、*无环性*、*不相交性*和*拥有关系*. *联系性* (connectedness) 意味着复合对象有一种包容性的坐标结构, 从这种对象的*起始地址* (starting address) 出发, 可以到达该对象的每个组分. *无环性* (noncircularity) 要求一个对象不能是其自身的一个子组分. 对象的*子组分* (subpart) 是指其组分, 或者其组分的子组分. (无环性蕴涵着一个对象不能是自己的组分.) *不相交性* (disjointness) 意味着如果两个对象有公共的子组分, 那么一定有一个对象是另一个的组分. *拥有关系* (ownership) 意味着拷贝一个对象就要拷贝它的组分, 而析构一个对象就要析构它的组分. 说一个对象是*动态的* (dynamic), 如果在它的生存期间其组分集合可能变化.

我们将称复合对象的类型为*复合对象类型*, 称由复合对象类型建模的概念为*复合对象概念*. 不可能针对*复合对象*定义任何算法, 因为复合对象是一个概

念模式而不是一个概念.

array_k 是概念 *Sequence* (序列) 的一个模型, 序列是一个复合对象概念, 它精化了 *Linearizable*, 它的组分就是它的元素范围:

$$
\begin{aligned}
&\mathit{Sequence}(\mathsf{S}) \triangleq \\
&\quad \mathit{Linearizable}(\mathsf{S}) \\
&\quad \wedge\ (\forall s \in \mathsf{S})\,(\forall i \in [\mathsf{begin}(s), \mathsf{end}(s)))\,\mathsf{deref}(i) \text{ 是 } s \text{ 的一部分} \\
&\quad \wedge\ = : \mathsf{S} \times \mathsf{S} \to \mathsf{bool} \\
&\qquad\qquad (s, s') \mapsto \mathsf{lexicographical\_equal}(\\
&\qquad\qquad\qquad\qquad \mathsf{begin}(s), \mathsf{end}(s), \mathsf{begin}(s'), \mathsf{end}(s')) \\
&\quad \wedge\ < : \mathsf{S} \times \mathsf{S} \to \mathsf{bool} \\
&\qquad\qquad (s, s') \mapsto \mathsf{lexicographical\_less}(\\
&\qquad\qquad\qquad\qquad \mathsf{begin}(s), \mathsf{end}(s), \mathsf{begin}(s'), \mathsf{end}(s'))
\end{aligned}
$$

如果 s 和 s' 相等但它们不是同一个序列, 那就一定有 $\mathsf{begin}(s) \neq \mathsf{begin}(s')$, 但 $\mathsf{source}(\mathsf{begin}(s)) = \mathsf{source}(\mathsf{begin}(s'))$. 这是**投影规范化** (projection regularity) 的一个实例. 请注意, 对于并非 *Sequence* 的 *Linearizable*, begin 和 end 也可以是规范的. 例如, 对于 bounded_range, 它们都是规范的.

练习 12.6 请定义性质 projection_regular_function.

12.2 动态序列

$\mathsf{array\_k}_{k,T}$ 是一个**常量规模的序列** (constant-size sequence). 参数 k 在编译时确定, 并将应用于这一类型的所有对象. 我们没有为常量规模的序列定义相应概念, 因为没有看到其他有用的模型. 类似的, 对于在构造时确定规模的序列, **固定规模的序列** (fixed-size sequence), 我们也不专门定义这种概念. 因为所有建模固定规模的序列的数据结构也都建模**动态规模的序列** (dynamic-size sequence). 这种动态序列的规模随元素插入删除而变化. (当然, 确实存在着一些固定规模的组合对象, 如 $n \times n$ 的方阵.)

无论建模动态序列的具体数据结构是什么, 对于规范类型的要求都支配着它的标准行为. 当它被销毁时, 其所有元素也都销毁, 这些元素占用的资源也将全部释放. 动态序列的相等和全序基于字典序定义, 就像 array_k 一样. 做一次

动态序列赋值, 左边得到的序列与右边的相等但不相交. 类似的, 拷贝构造函数也创建出相等的但不相交的序列.

如果 s 是动态规模的 (或简称*动态的*, dynamic) 序列, 其规模是 $n \geqslant 0$, 在*插入索引* (insertion index) i 处*插入* (inserting) 一个规模为 k 的范围 r, 将使其规模增加到 $n+k$. 插入索引 i 可以是闭区间 $[0,n]$ 的 $n+1$ 个值里的任何一个. 如果 s' 是插入后序列的值, 那么

$$s'[j] = \begin{cases} s[j] & \text{if } 0 \leqslant j < i \\ r[j-i] & \text{if } i \leqslant j < i+k \\ s[j-k] & \text{if } i+k \leqslant j < n+k \end{cases}$$

类似的, 如果 s 是一个规模为 $n \geqslant k$ 的序列, 在其*删除索引* (erasure index) i 处*删除* (erasing) k 个元素, 将使其规模减少到 $n-k$. 删除索引 i 可以是闭区间 $[0,n-k]$ 里 $n-k+1$ 个值中的任何一个. 如果 s' 是删除后序列的值, 那么

$$s'[j] = \begin{cases} s[j] & \text{if } 0 \leqslant j < i \\ s[j+k] & \text{if } i \leqslant j < n-k \end{cases}$$

对于插入和删除的不同需要, 引出了顺序型数据结构的许多变形, 它们有着不同的插入删除复杂性. 不同类别主要看相应的结构是否存在*远程* (remote) 组分. 对于一个对象的一个组分, 如果它并不驻留在从该对象的地址出发的某个常量偏移位置, 只能通过从对象*头部*开始, 通过遍历该对象的坐标结构的方式才能达到, 这一组分就称为是远程的. 复合对象的头部是指它的所有*局部* (local) 组分的汇集. 局部组分即是那些驻留在从对象的起始地址开始具有常量偏移位置的组分. 一个对象的局部组分的量是由其类型确定的一个常数.

本节将总结顺序性数据结构的一些性质. 我们把这些数据结构分为两个不同的基本类别: *链接的* (linked) 和*基于分区的* (extent-based).

*链接*数据结构通过用于链接的指针联系起有关的数据组分. 每个元素驻留在一个独立的*具有固定定位的* (permanently placed) 组分里. 所谓具有固定定位, 是指在该元素生存期间其地址绝不变化. 除了这个元素之外, 该组分还包含与相邻组分的连接器. 这里的迭代器是链接迭代器, 它不支持索引迭代器. 插入和删除可能只需常量时间, 因为它们都可以通过重新链接的方式完成, 因此不会让迭代器失效. 链接表有两种主要变体: 单链接的和双链接的.

一个**单链接表** (singly linked list) 只有一个链接的 *ForwardIterator*. 在特定的迭代器之后插入删除的代价是常量的, 而在任意迭代器之前插入或删除, 代价正比于从表头到这里的距离. 这样, 在表头插入和删除的代价总是常量. 单链表也有几种变体, 它们之间的差异在于头部的结构, 以及是否存在到最后元素的链接. 一个**基本**单链表的头部只有一个到表中首元素的链接, 它也可能取特殊的空 (null) 值表明这是一个空表. 没有到最后元素的链接. **循环** (circular) 单链表的头部只包含到表中最后一个元素的链接, 它也可能是空以表示空表; 这里最后元素的链接指向第一个元素. 一个**首尾表** (first-last list) 的头部由两个部分组成: 一个指向由空结束的基本表的第一个元素, 还有一个是到最后元素的链接. 如果表空则后者也取空值.

有几个因素影响到对单链表实现方式的选择. 如果在一个应用里存在着大量的表, 而且其中很多都是空的, 那么尽可能小的头部就有价值. 循环表的迭代器可能比较大, 而且其 successor 操作较慢, 因为这里必须区分指向第一个元素和指向表后端极限的指针. 如果一个数据结构只需常量时间就可以完成尾部插入, 它就可以用于实现队列或者输出受限的双端队列. 下表总结了不同的表实现上的各种权衡情况:

变形	一个字的头部	简单的迭代器	后端插入
基本	是	是	否
循环	是	否	是
首尾	否	是	是

双链接表具有链接的*BidirectionalIterator* (双向迭代器). 插入 (在一个迭代器前后) 和删除的代价都是常量. 与单链表一样, 双链表也有几种变形. **循环双链表**的头部包含一个指针, 指向表中第一个元素, 或者为空表示这是一个空表. 第一个元素的反向指针指向最后一个元素, 而最后一个元素的前向指针指向第一个元素. **带哑结点** (dummy node) 的表与循环表类似, 但在最后元素和第一个元素之间另有一个哑结点, 头部包含的是到该哑结点的链接. 在哑结点里可能没有实际数据对象. **双指针头部** (two-pointer header) 双链表的情况与哑结点表类似, 但头部包含两个指针, 其值对应于哑结点的两个链接.

如前所述, 有两个因素影响到人们在不同的单链表实现中的选择, 也就是说头部的规模和迭代器复杂性, 它们也都与在不同双链表实现中的选择有关.

但双链表还有些特殊的问题. 如果表有一个固定的极限迭代器, 有些算法就可能简化, 因为这一极限可以当作与在整个表上的任何合法迭代器都不同的值. 正如将在本章后面看到的, 如果存在从远程组分到局部组分的链接, 也会大大增加在该种类型的表里做元素重整的代价. 下表总结了这些实现权衡:

变体	一个字的头部	简单迭代器	无远程到局部的链接	永久的极限
循环	是	否	是	否
哑结点	是	是	是	否[3]
双指针	否	是	否	是

第 8 章介绍过*链接重整* (link rearrangement), 其中只是重新安排了一个或几个链接范围里的链接迭代器之间的连接关系, 并不创建或销毁迭代器, 也不改变迭代器与它们指向的对象之间的关系. 链接重整可以限制在一个表的内部, 也可以涉及多个表, 从而导致元素拥有关系的变化. 举例说, split_linked 用于把满足谓词的元素从一个表移到另一个表, 而 combine_linked_nonempty 用于移出一个表里的元素, 并将它们归并到另一个表. *粘接* (splicing) 也是一种链接重整, 它从一个表中删除一个范围, 并将它重新插入到另一个表里.

在一些算法里没使用链接结构里的反向链接, 例如做排序的算法. 但这种链接却很有用, 它使在任意位置的插入或删除都能在常量时间完成, 而在单链表里做类似操作的代价可能高得多. 由于插入和删除的代价经常是选用链接表的首要原因, 因此实际中确实应该认真考虑双链表.

基于分区的 (extent-based) 数据结构把元素分组存入一个或几个分区 (extent), 或者存入远程的数据组分的区块, 并支持对它们的随机访问. 在任意位置插入删除所需的时间与序列的规模成正比, 而在最后 (或可能还包括最前) 的插入和删除只需要分期付款式的常量时间.[4] 对于每种实现, 插入和删除都会按确定的规则使一些特定迭代器非法; 换句话说, 这里的任何元素都没有持

---

3. 如果即使表为空时也分配一个哑结点, 那么就有了一个持续存在的极限; 不幸的是, 这种做法违背了我们一直希望有的一种性质: 空数据结构里不应该有任何远程组分, 这样才能保证不使用任何附加资源就可以构造相关的数据结构.

4. 一个运算的*分期付款复杂性* (amortized complexity) 就是最坏情况下的一系列运算的平均复杂性. 分期付款复杂性的概念由 Tarjan [1985] 引入.

久确定的位置. 一些基于分区的数据结构使用*单分区* (single extent), 另一些是*分段的* (segmented), 其中使用多个分区和一些附加的索引结构.

在单分区数组里, 有关分区只需要在规模非 0 时存在. 为避免每次插入时重新分配, 分区里应该保留一部分空间; 只有在保留空间用完时才重新分配分区. 头部需包含一个到分区的指针; 还需要额外的指针维护数据和保留空间的轨迹, 数据元素通常保存在分区的前部. 如果存在嵌套的数组, 在分区前部 (而不是在头部中) 放这些指针, 可以改善空间和时间复杂性.

单分区数组也有几种变形. 在*单端* (single-ended) 数组里, 数据从分区中某个固定的偏移值处开始存放, 数据之后是保留空间.[5] 在*双端* (double-ended) 数组里, 数据位于分区的中间部分, 其两边都有保留空间. 如果任何一端的保留空间耗尽, 就重新分配这一分区. 在*循环* (circular) 数组里, 分区的处理方式就像是其最高地址的后面就是最低地址. 这样, 从逻辑上看, 一块保留空间就像是既在数据之前又在其之后, 可以向两个方向增长.

有几个因素影响到单分区数组的实现选择. 对单端和双端数组, 直接采用机器地址是迭代器的最高效实现方式; 循环数组的迭代器规模大一些, 而且其遍历函数也慢一些, 因为这里需要维持有关的轨迹, 确定正在使用的空间是否卷回到分区的开始. 如果一种数据结构还能支持前端的常量时间插入和删除, 它就可以用作队列或者输出受限的*双端队列* (deque). 在任何一端用尽了, 即使另一端还有可用空间, 双端数组都需要重新分配. 而对单端的或循环的数组, 可以到保留空间耗尽时再重新分配.

变体	简单迭代器	前端插入/删除	重新分配的有效性
单端	是	否	是
双端	是	是	否
循环	否	是	是

当一次插入遇到了单端或循环数组的分区满时, 就会发生*重新分配* (reallocation): 分配一块更大的分区, 并把现存元素移到新分区中. 对于双端数组, 在数组中耗尽的一端插入时需要重新分配, 也可以把元素向数组的另一端

5. 当然, 完全可以让数据从后面向前反向排列, 但是我们没看到这种做法的实际意义.

移动, 重新安排剩下的保留区. 无论是做重新分配, 还是在双端数组里移动元素, 都会导致指到数组里面的所有迭代器不再合法.

在需要重新分配时, 按乘以一个因子的方式增加分区的规模, 可以得到按每个元素计算的分期付款的常量代价. 根据我们的经验, 以 2 作为增长的因子, 可以在尽可能减少每个元素的平均分期付款代价, 与较好的存储利用率之间取得很好的平衡.

练习 12.7 请推导出相应的表达式, 说明对不同的因子的存储利用率和按每个元素计算的重新构造次数.

项目 12.1 请结合理论分析和实际试验, 确定在各种负载下, 对于单分区数组的最优重分配策略.

对于单端的或循环的单分区数组 a, 可以定义一个函数 capacity (容量), 使 size(a) ⩽ capacity(a), 只有在发现执行插入操作后就会超出 a 的容量时, 才先执行重新分配后再插入. 还可以定义过程 reserve (保留), 允许把数组的容量提升到某个指定量.

练习 12.8 请为双端数组的容量和保留操作设计接口.

一个*分段的* (segmented) 数组包含一块或几块保存元素的分区, 还有一个*索引* (index) 数据结构管理指向这些分区的指针. 由于需要检测是否到达分区结束, 这里的迭代器遍历函数将比单分区数组的情况慢一些. 索引结构应表现出与分段数组一样的行为: 要支持随机访问、后端插入和删除. 如果需要, 还应支持相应的前端操作. 这里不需要做完全的重新分配, 因为在一个现存的分区满时总可以再加入一个分区. 只需在全部分区的一端或两端保留空间.

分段数组的主要变化是索引的结构. *单分区* (single-extent) 索引本身是一个单分区数组, *分段* (segmented) 索引本身又是一个分段数组, 通常它还有一个单分区索引, 也有可能有进一步的分段索引. *斜形* (slanted) 索引可以有很多层, 其根是一个固定规模的分区; 前面几个元素是指向数据分区的指针; 下一个指针指向一个间接的索引分区, 其中是指向数据分区的指针; 再后面的指针指向一个双重间接的分区, 里面是指向间接索引分区的指针; 如此等等.[6]

6. 这一设计基于 UNIX 的文件系统 [参见 Thompson and Ritchie 1974].

项目 12.2 请为动态序列设计一套完整接口. 应包含构造, 插入, 删除, 和拼接 (splicing). 要保证存在不同的变体, 以处理不同实现方式的一些特殊情况. 例如, 不仅要能在指定的迭代器之前插入, 还应能在迭代器之后插入, 以便能用于单链表.

项目 12.3 请实现一个综合性的动态序列库, 提供各种单链表, 双链表, 单分区和分段的数据结构.

项目 12.4 请基于实际应用的负荷情况为动态序列设计一个标准测试集, 以便度量各种数据结构的性能. 进而基于上述工作的结果为用户提供一个选择指南.

12.3 基础类型

第 2 到 5 章研究了各种数学值上的算法, 它们说明了规范类型支持的等值推理不仅可以用于算法, 也可以用于证明. 第 6 到 11 章研究存储区上的算法, 从中可以看到, 等值推理在有变化的状态的世界里依然很有用. 但是, 那里处理的都是小对象, 例如整数和指针, 其特点就是赋值或拷贝的代价很低. 本章介绍了复合对象的概念, 它们也满足规范类型的要求, 因此又可以用作其他复合对象的元素. 由于在动态序列和其他一些复合对象里区分了头部和远程组分, 因此有可能实现重整的高效方法: 只需移动头部而不移动远程组分.

为了理解导致复合对象的重整比较低效的根源, 考虑下面 swap_basic 过程:

```
template<typename T>
    requires(Regular(T))
void swap_basic(T& x, T& y)
{
    T tmp = x;
    x = y;
    y = tmp;
}
```

假设现在调用 swap_basic(a, b) 去交换两个动态序列, 易见, 它执行的构造和两个赋值都需要线性时间. 进一步说, 虽然这里原本不需要增加任何存储,

但却可能导致内存耗尽异常.

避免这种高代价拷贝的方法是采用一种专门化的 swap_basic, 它只交换特定动态序列类型的对象的头部, 如果需要就更新头部里到远程组分的指针. 然而, 这种专门化的 swap_basic 也有问题. 首先, 需要为每个数据结构重复提供这一操作. 更重要的是, 许多重整算法并不是基于 swap_basic 定义的, 包括各种原地置换算法, 如 cycle_from, 以及各种使用缓冲区的算法, 如 merge_n_with_buffer. 最后, 还存在另外一些情况, 例如在单分区数组重新分配时, 需要把老分区里的对象移到一个新分区.

这里的想法是把交换头部的思想推广到任意的重整, 希望能允许使用缓冲区或进行重新分配, 并希望继续写抽象算法, 保证它们不依赖于被操作对象的实现. 为达到这一目标, 我们要给每个规范类型 T 关联它的基础类型 (underlying type), U = UnderlyingType(T). 如果类型 T 没有远程组分, 或者其远程组分中有回到头部的链接, 那么其基础类型 U 与 T 相同.[7] 否则, U 在所有方面都和 T 一样, 除了它不维持 T 的拥有关系: 构造时不影响相关的远程组分, 也就是说, 拷贝构造和赋值时都只拷贝头部, 不拷贝远程组分. 当基础类型与原类型不同时, 它具有与原类型的头部相同的布局 (相同的二进制模式).

具有同样二进制模式这一事实可以有下面解释: 一个类型与其基础类型的对象间的这种关系, 使人可以通过语言内部的 reinterpret_cast 函数模板, 按这种或那种观点去看这片内存. 在实现类型 T 对象的重整时, 只能用 UnderlyingType(T) 对象保存临时值. 对真的 (proper) 基础类型 (即, 它不等于原类型), 其拷贝构造和赋值的复杂性正比于类型 T 的头部的规模. 这种情况还有另一个重要收获: UnderlyingType(T) 的拷贝构造和赋值绝不会抛出异常.

原类型 T 的基础类型的实现是直截了当的, 而且可能自动化. U = UnderlyingType(T) 总具有和 T 的头部相同的布局. 这样, U 的拷贝构造和赋值也就是拷贝这些二进制位, 并不构造 T 的远程组分的拷贝. 例如, pair_{T_0,T_1} 的基础类型就是一个二元组, 其成员就是 T_0 和 T_1 的基础类型. 其他类型的情况也都类似. $\text{array\_k}_{k,T}$ 的基础类型是数组 array_k_k, 其元素是 T 的基础类型.

一旦定义了 UnderlyingType(T), 就可以用下面过程把对 T 引用强制到 UnderlyingType(T), 这里不需要做任何计算:

---

7. 这一解释也是对从远程组分到头部的链接的警告, 如讨论双链表所说的.

```
template<typename T>
    requires(Regular(T))
UnderlyingType(T)& underlying_ref(T& x)
{
    return reinterpret_cast<UnderlyingType(T)&>(x);
}
```

现在可以把 swap_basic 重写为下面样子, 从而高效地交换两个复合对象:

```
template<typename T>
    requires(Regular(T))
void swap(T& x, T& y)
{
    UnderlyingType(T) tmp = underlying_ref(x);
    underlying_ref(x)     = underlying_ref(y);
    underlying_ref(y)     = tmp;
}
```

同样工作也可以用下面方式完成:

```
swap_basic(underlying_ref(x), underlying_ref(y));
```

只需简单地按重新实现 swap 的同样方法重新实现 exchange_values, 就可以把许多重整算法改变为使用基础类型了.

要处理其他重整算法, 可以用一个迭代器适配器. 这种适配器具有与原迭代器相同的遍历操作, 但其值类型换成了原来值类型的基础类型. 进一步的, 让 source 返回 underlying_ref(source(x.i)), 让 sink 返回 underlying_ref(sink(x.i)). 这里的 x 是适配器对象, 而 i 是在 x 里的原迭代器对象.

练习 12.9 请实现这样的适配器, 使之能用于所有迭代器概念.

现在可以把 reverse_n_with_temporary_buffer 重新实现为:

```
template<typename I>
    requires(Mutable(I) && ForwardIterator(I))
void reverse_n_with_temporary_buffer(I f, DistanceType(I) n)
```

```
{
    // 前条件: mutable_counted_range(f,n)
    temporary_buffer<UnderlyingType(ValueType(I))> b(n);
    reverse_n_adaptive(underlying_iterator<I>(f), n,
                       begin(b), size(b));
}
```

这里的 underlying_iterator 是取自练习 12.9 的适配器.

项目 12.5 用基础类型系统地彻底改造一个主要的 C++ 库, 例如 STL, 或者基于本书的思想实现一个新的库.

12.4 总结

通过扩展 C++ 的结构类型和常量规模的数组类型, 可以得到带有远程部分的动态数据结构. 拥有关系和规范性的概念决定了拷贝构造、赋值、相等和全序运算中对组分的处理. 本章随后以动态序列作为实例, 说明了各种数据结构的各种有用变形都需要仔细实现, 并需要写出文档, 使程序员能够为每个具体应用选出最合适的结构. 这里还阐释了, 通过临时性地放松拥有关系不变式, 可以高效实现嵌套数据结构的各种重整.

跋

我们要在此回顾本书的各个主题: 规范性、概念、算法及其接口、编程技术, 以及指针的意义. 还要讨论每个主题的特殊的局限性.

规范性

规范类型采用与相等关系一致的方式定义拷贝构造和赋值操作. 规范函数对相等的输入返回相等的输出. 作为例子, 规范变换使我们可以定义分析轨道的算法, 并对这种算法做推理. 贯穿本书的序关系、前向迭代器的后继关系, 以及许多其他东西, 都以规范性作为基础.

用语言内部的类型工作时, 通常都把判断相等、拷贝和赋值操作的复杂性看作是常量的. 在处理复合对象时, 也期望这些操作的复杂性与对象的面积 (area, 也就是其内存占用的总量, 包括其局部组分和远程组分) 成线性关系, 期望在最坏情况时, 相等判断的代价也与其参数的空间成线性关系. 但是, 实际情况中这些期望却未必都能满足.

举个例子, 如果将多重集 (multiset, 即可能包含重复元素的无序汇集) 表示为不排序的动态序列. 虽然插入新元素只需要常量时间, 但检测两个多重集相等却需要 $O(n \log n)$ 时间: 先对两个多重集排序, 而后再按字典序比较. 如果不经常做相等检测, 这样处理就很合适; 但是, 如果需要把一批这样的多重集放进一个序列, 而后用 find 检索, 操作的性能将是无法接受的. 作为另一个极端的例子, 考虑一个情况, 其中某个类型的相等必须通过图同构判断来实现, 而图同构问题是一个著名的不存在多项式时间算法的问题.

第 1.2 节指出, 如果实现值上的行为相等在实际中不可行,我们经常会考虑用表示相等来代替之. 对复合对象, 经常用第 7.4 节的技术来实现表示相等. 这种结构相等 (structural equality) 在定义拷贝构造和赋值的语义时很有用, 也常

常有助于处理其他问题. 请回忆一下, 表示相等蕴涵着行为相等. 与此类似, 在无法实现一种自然的全序时, 一种基于结构的默认全序 (例如序列的字典序) 使我们能有效实现排序和检索. 当然, 也存在一些情况, 其中的对象没有拷贝构造也没有赋值, 甚至没有相等判断, 因为它们拥有的资源是唯一的.

概念

前面用了一些取自抽象代数的概念 (如半群、幺半群和模) 来描述各种算法, 例如乘方、余数和 gcd 等. 许多情况下需要对标准的数学概念做一点调整, 使之适合算法的需要. 有时还需引进新概念来强化需求, 例如 *HalvableMonoid*. 有时放松了一些需求, 例如在考虑 partially_associative 时. 我们经常需要处理部分的定义域, 例如把定义空间谓词传给 collision_point. 数学概念是可以使用和自由修改的工具, 出自计算机科学的概念也一样. 迭代器的概念描述了一些特定算法和数据结构的基本性质; 当然, 描述其他坐标结构的概念还有待进一步开发. 程序员的一项工作就是确定某个特定的概念是否有用.

算法及其接口

有界的半开范围可以很自然的对应到许多数据结构的实现, 它提供了一种表示许多算法的输入和输出的方便方式, 例如查找、旋转、划分、归并等等. 然而, 对有些算法, 例如 partition_point_n, 更自然的接口是计数范围. 即使对那些可以很自然地以有界范围作为接口的算法, 通常也存在它们的以计数范围为接口的变体. 将自己限制到单一的接口不是最经济的工作方式.

第 10 章描述的三个旋转算法对应于三种不同的迭代器类型. 对每一个算法, 都需要去发现它的概念需求. 在它的输入上的前条件, 以及适合使用它的其他特性. 单一算法能适应于所有使用的情景是很罕见的.

编程技术

successor 是一个纯函数式的变换, 基于它可以写出许许多多清晰而高效的算法. 然而在第 9 章, 我们却把对 successor 和 predecessor 的调用封装到很小的变动性机器里, 例如 copy_step, 因为这样做对一族相关算法都可以得到更清晰的代码. 类似的, 第 8 章在状态机器里使用 goto, 第 12 章为基础类型机制使用

reinterpret_cast 都很合适. 我们不应该去限制基础机器和语言的表达能力, 而应该为每种可用结构确定适当的用法. 好的软件产品出自各种组件的正确组织, 而不是出自一套语法的或者语义的约束.

指针的意义

本书阐释了使用指针的两种方式: (1) 作为迭代器或者其他坐标, 在算法里表示数据结构中的位置; (2) 作为 *连接器* (connector), 表示一个复合对象对其远程部分的拥有关系. 例如, 第 12.2 节讨论了用指针连接一个表里的结点, 以及用指针连接一个数组的一批分区.

指针的这两种用途也确定了在一个对象被拷贝、销毁或比较相等时的不同行为. 拷贝一个对象时需要沿着其连接器拷贝远程组分, 所以新对象将包含新的连接器, 它们指向那些被拷贝的新组分. 另一方面, 拷贝包含迭代器的对象时 (例如对一个 bounded_range), 则只应简单地拷贝迭代器而不要追溯它们去继续拷贝. 与此类似, 销毁一个对象时需要沿着它的连接器销毁其远程组分, 而销毁一个包含迭代器的对象则不影响被迭代器指向的那些对象. 最后, 比较容器的相等需要沿着连接器去比较对应的组分, 而非容器的相等 (例如对于 bounded_range) 则只应检测迭代器相等.

当然, 还存在指针的第三种使用: 表示事物之间的 *关系* (relationship). 两个或更多对象之间的关系并不是这些对象所拥有的组分; 一个关系有它自己的存在, 同时又维持它所联系的对象之间的相互依赖. 一般而言, 表示关系的指针并不参与各种常规运算. 例如, 拷贝一个对象时不需要检查或拷贝关系指针, 因为相应关系只对这一对象有效, 与其拷贝无关. 如果一个一对一关系用一对链接两个对象的嵌入式的指针表示, 无论销毁哪个对象, 都需要清除位于另一对象里的相应指针.

在将数据结构设计为带有拥有关系和远程组分的复合对象时, 应该采用这样一种编程风格, 其中让各种基本对象 (即那些不作为其他对象的子部分的对象) 都驻留在静态变量里, 其生存期是程序的整个执行期间 (对于局部变量, 其生存期是一个分程序的执行期间), 而让动态分配的部分仅作为远程组分. 这样就把 Algol 60 的基于栈的块结构拓展到处理任意的数据结构. 这种结构能自然地符合许多应用的需要. 当然, 也存在一些情形, 其中需要采用引用计数、废料收集或其他存储管理技术.

总结

程序设计是一种迭代式过程: 研究有用的问题, 发现处理它们的高效算法, 精炼出算法背后的概念, 再将这些概念和算法组织为完满协调的数学理论. 每一个新发现都带来知识的增长, 但每个知识片段又有其自身的局限性.

附录 A

数学表示

我们用符号 $\triangleq$ 表示 “按定义相等”.

如果 P 和 Q 是命题, 那么 $\neg P$ (读作 “非 P”)、 $P \vee Q$ (“P 或 Q”)、 $P \wedge Q$ (“P 与 Q”)、 $P \Rightarrow Q$ (“P 蕴涵 Q”) 以及 $P \Leftrightarrow Q$ (“P 等价于 Q”), 也都是命题. 对于等价, 也经常说 “P 当且仅当 Q”.

如果 P 是命题而 x 是变量, 那么 $(\exists x)P$ 也是一个命题 (读作 “存在 x 使得 P”). 如果 P 是命题而 x 是变量, 那么 $(\forall x)P$ 也是一个命题 (读作 “对所有的 x, P”); $(\forall x)P \Leftrightarrow (\neg(\exists x)\neg P)$.

本书使用了下面来自集合论的术语和记法:

$a \in X$ (“a 是 X 的元素”)

$X \subset Y$ (“X 是 Y 的子集”)

$\{a_0, \ldots, a_n\}$ (“由元素 $a_0, \ldots, a_n$ 组成的有穷集”)

$\{a \in X | P(a)\}$ (“X 中满足谓词 P 的所有元素构成的子集”)

$X \cup Y$ (“X 和 Y 的并集”)

$X \cap Y$ (“X 和 Y 的交集”)

$X \times Y$ (“X 和 Y 的直积”)

$f : X \to Y$ (“f 是从 X 到 Y 的函数”)

$f : X_0 \times X_1 \to Y$ (“f 是从 X_0 和 X_1 的乘积到 Y 的函数”)

$x \mapsto \mathcal{E}(x)$ (“x 映射到 $\mathcal{E}(x)$”, 总写在给了函数签名之后)

闭区间 (closed interval) $[a, b]$ 是所有满足 $a \leqslant x \leqslant b$ 的元素 x 的集合. 开区间 (open interval) (a, b) 是所有满足 $a < x < b$ 的元素 x 的集合. 右半开区间 (half-open-on-right interval) $[a, b)$ 是所有满足 $a \leqslant x < b$ 的元素 x 的集合. 左半开区间 (half-open-on-left interval) $(a, b]$ 是所有满足 $a < x \leqslant b$ 的元素 x 的集合.

半开区间 (half-open interval) 是右半开区间的简称. 这些定义都推广到弱序.

规程 (specification) 中使用了下面记法, 其中的 i 和 j 是迭代器, n 是整数:

$i \prec j$ ("i 先于 j")

$i \preceq j$ ("i 先于或等于 j")

$[i, j)$ ("从 i 到 j 的半开有界范围")

$[i, j]$ ("从 i 到 j 的闭有界范围")

$[\![i, n)\!)$ ("从 i 开始 $n \geqslant 0$ 的半开的弱或计数范围")

$[\![i, n]\!]$ ("从 i 开始 $n \geqslant 0$ 的闭的弱或计数范围")

讨论概念时采用了下面术语:

弱 (weak) 表示要求减弱公理, 包括放松一些要求. 如弱序用等价代替相等.

半 (semi) 指减少一个运算. 例如, 半群没有求逆.

部分 (partial) 指对定义空间的限制. 例如, 部分减法 (partial subtraction, 或称删除, cancellation) $a - b$ 只在 $a \geqslant b$ 时有定义.

附录 B

程序设计语言

Sean Parent 和 Bjarne Stroustrup

本附录定义书中使用的 C++ 子集. 为简化语法形式, 这里把一些库功能当作内部功能使用. 这些功能没有用这一子集书写, 其中用到一些另外的 C++ 特征. B.1 节定义这一 C++ 子集; B.2 节描述上述内部功能的实现.

B.1 语言定义

语法记法约定

这里采用的是 Niklaus Wirth 设计的一种 Backus-Naur 范式 (Backus-Naur Form) 的扩充形式. Wirth [1977, 822–823 页] 对这种形式的说明如下:

> 术语标识符 (identifier) 用于指非终结符号 (nonterminal symbol), 而文字 (literal) 指终结符号 (terminal symbol). 为简洁起见, identifier和character的进一步细节都没有定义.

```
syntax     = {production}.
production = identifier "=" expression ".".
expression = term {"|" term}.
term       = factor {factor}.
factor     = identifier | literal
               | "(" expression ")"
               | "[" expression "]"
               | "{" expression "}".
```

```
literal     = """" character {character} """".
```

重复用花括号表示, 也就是说, {a} 表示 ϵ | a | aa | aaa | ···. 可选项用方括号, 这样 [a] 表示 a | ϵ . 圆括号只用于分组, 例如 (a | b) c 表示 ac | bc. 终结符号 (即文字) 用双引号括起 (鉴于此, 如果引号本身作为文字出现, 就将它写两次).

词法约定

下面产生式给出了标识符和文字的语法:

```
identifer = (letter | "_") {letter | "_" | digit}.
literal   = boolean | integer | real.
boolean   = "false" | "true".
integer   = digit {digit}.
real      = integer "." [integer] | "." integer.
```

注释从两个斜线开始直至一行结束:

```
comment     = "//" {character} eol.
```

基本类型

这里用到 C++ 的三个类型: bool 只包含值 false 和 true, int 包含带符号的整数值, 还有 double 是 IEEE 64 位标准的浮点值:

```
basic_type = "bool" | "int" | "double".
```

表达式

表达式可以有运行时的表达式或编译时的表达式. 编译时表达式可能在编译中求值出一个值或一个类型.

表达式由下面语法定义. 内层产生式里的运算符 (在语法里写的靠下) 的优先级高于外层产生式里的运算符:

```
expression      = conjunction {"||" conjunction}.
conjunction     = equality {"&&" equality}.
equality        = relational {("==" | "!=") relational}.
relational      = additive {("<" | ">" | "<=" | ">=") additive}.
additive        = multiplicative {("+" | "-") multiplicative}.
multiplicative  = prefix {("*" | "/" | "%") prefix}.
prefix          = ["-" | "!" | "const"] postfix.
postfix         = primary {"." identifier
                    | "(" [expression_list] ")"
                    | "[" expression "]"
                    | "&"}.
primary         = literal | identifier | "(" expression ")"
                    | basic_type | template_name | "typename".

expression_list = expression {"," expression}.
```

|| 和 && 运算符分别表示 ∨ (析取) 和 ∧ (合取). 它们的运算对象必须具有布尔值. 其第一个运算对象先于第二个求值. 如果根据第一个运算对象的值足以确定表达式的结果 (对于 || 是 true, 或对于 && 是 false), 第二个运算对象就不再求值, 结果就是第一个运算对象的值. 前缀运算符 ! 是 ¬ (否定), 只能用于布尔值.

== 和 != 分别表示相等和不等运算符, 它们返回布尔值.

<、>、<= 和 >= 分别表示小于、大于、小于等于和大于等于, 它们都返回布尔值.

+ 和 - 分别是加和减, 一元前缀 - 是求负.

*、/ 和 % 分别是乘、除和求余数.

后缀 . (圆点) 的左边应该是一个结构类型的对象, 它返回跟随圆点之后的标识符表示的成员. 后缀 () 的左边应是一个过程或者一个应用运算符有定义的对象, 返回对于给定的实参应用这个过程或者函数对象的结果. 如果是应用到一个类型, () 用给定参数做相应的对象构造, 如果应用到一个类型函数, 它返回另一个类型. 后缀 [] 的左边应是一个索引运算符有定义的对象, 它返回由方括号里的表达式的值确定位置的那个元素.

前缀 const 是一个类型运算符, 返回其参数类型的常量类型. 在用于引用类型时, 得到的是其基础引用类型的一个常量版本的类型.

后缀 & 是一个类型运算符, 返回其参数类型的引用类型.

枚举

一个枚举生成一个类型, 对应于表中的每个标识符有唯一的一个值. 枚举上有定义的仅有运算就是规范类型上都有的几个: 相等、关系运算、不等、构造、析构和赋值:

```
enumeration    = "enum" identifier "{" identifer_list "}" ";".
identifer_list = identifier {"," identifier}.
```

结构

一个结构是一个类型, 它由一组有类型的对象 (称为其数据成员) 组成, 这些对象采用互不相同的命名. 某个数据成员或是一个独立的对象, 或是常量大小的数组. 此外, 一个结构还可能包含一些构造函数、一个析构函数和一些成员运算符 (赋值、应用运算符和索引) 的定义, 还可以包含一些局部的 typedef. 带有应用运算符成员的结构称为*函数对象* (function object). 去掉结构体 (structure_body) 就是结构的事先声明.

```
structure        = "struct" structure_name [structure_body] ";".
structure_name   = identifier.
structure_body   = "{" {member} "}".
member           = data_member
                   | constructor | destructor
                   | assign | apply | index
                   | typedef.
data_member      = expression identifier ["[" expression "]"] ";".
constructor      = structure_name "(" [parameter_list] ")"
                   [":" initializer_list] body.
destructor       = "~" structure_name "(" ")" body.
```

```
assign          = "void" "operator" "="
                    "(" parameter ")" body.
apply           = expression "operator" "(" ")"
                    "(" [parameter_list] ")" body.
index           = expression "operator" "[" "]"
                    "(" parameter ")" body.

initializer_list = initializer {"," initializer}.
initializer      = identifer "(" [expression_list] ")".
```

以一个到本类结构的常量应用为参数的构造函数称为*拷贝构造函数* (copy constructor). 如果没有定义拷贝构造函数, 编译器自动生成一个按成员逐个做的拷贝构造函数. 没有参数的构造函数称为*默认构造函数* (default constructor). 如果没定义任何构造函数, 编译器自动生成一个默认构造函数, 它一个个地构造各个成员. 如果没有定义赋值运算符, 也将自动生成一个按成员拷贝的赋值运算符. 初始化表 (initializer list) 里的每个标识符应是该结构的一个数据成员的标识符. 如果一个构造函数包含初始化表, 表中的数据成员将用与相应初始式 (initializer) 匹配的构造函数进行构造[1]; 所有这些构造都发生在构造函数的体执行之前.

过程

一个过程包含一个返回类型 (没有返回值时用 void), 随后是它的名字和形参表. 名字可以是标识符或者运算符. 形参表里的 expression 必须能说明一个类型. 可以用不包括体的过程签名作为事先声明.

```
procedure      = (expression | "void") procedure_name
                   "(" [parameter_list] ")" (body | ";").
procedure_name = identifier | operator.
operator       = "operator"
                   ("==" | "<" | "+" | "-" | "*" | "/" | "%").
```

1. 执行重载解析时所用的匹配机制只做准确匹配, 不做任何隐式转换.

```
parameter_list = parameter {"," parameter}.
parameter      = expression [identifier].
body           = compound.
```

只有这里列出的运算符可以定义. 对运算符 != 的定义基于 == 生成; 对运算符 >、<= 和 >= 的定义基于 < 生成. 在调用一个过程时, 各个实参表达式的值绑定到对应的形参, 而后执行过程体.

语句

过程、构造函数、析构函数和成员运算符的体是语句:

```
statement         = [identifier ":"]
                      (simple_statement | assignment
                      | construction | control_statement
                      | typedef).
simple_statement  = expression ";".
assignment        = expression "=" expression ";".
construction      = expression identifier [initialization] ";".
initialization    = "(" expression_list ")" | "=" expression.
control_statement = return | conditional | switch | while | do
                      | compound | break | goto.
return            = "return" [expression] ";".
conditional       = "if" "(" expression ")" statement
                      ["else" statement].
switch            = "switch" "(" expression ")" "{" {case} "}".
case              = "case" expression ":" {statement}.
while             = "while" "(" expression ")" statement.
do                = "do" statement
                      "while" "(" expression ")" ";".
compound          = "{" {statement} "}".
break             = "break" ";".
goto              = "goto" identifier ";".
```

```
typedef              = "typedef" expression identifier ";".
```

对简单语句 (常见的如过程调用) 求值是为了它的副作用. 赋值 (assignment) 也就是应用其左边的那个对象的类型的赋值运算符. 构造 (construction) 的 expression 是一个类型表达式, 给出被构造的类型. 没有初始化部分 (initialization) 而构造时将应用默认构造函数. 带有括起的表达式的构造应用与之匹配的构造函数. 如果是用一个等号构造, 跟随其后的就是送给拷贝构造函数的表达式, 该表达式必须具有与所构造对象同样的类型.

return 语句将控制返回当前函数的调用方, 还返回表达式的值作为函数结果. 该表达式必须能求出函数的返回值类型的值.

如果条件语句 (conditional) 的 expression 的值是真, 它就执行其第一个语句, 如果是假而且有 else 子句, 就执行第二个语句. 这里的 expression 必须求出布尔值.

switch 语句先求值 expression, 而后执行与得到的值匹配的第一个 case 标号之后的语句; 然后执行随后的语句, 直至 switch 语句结束, 或者直至执行到一个 break 语句. switch 语句的 expression 必须求出一个整数或者枚举值.

while 语句反复求值其 expression, 只要它为真就执行 statement. do 语句反复执行其 statement 而后求值其 expression, 直至该 expression 的值为假. 对于这两种情况, expression 都必须求出布尔值.

复合语句 (compound) 按顺序执行其语句序列.

goto 语句使执行转到当前过程中对应标号处的语句.

break 语句终止外围最小的 switch、while 或者 do 语句, 使执行转到被终止的语句之后的语句并从那里继续.

typedef 语句为一个类型定义别名.

模板

模板使我们可以用一个或多个类型或常量将结构或过程参数化. 模板定义和模板名用 < 和 > 作为分隔符.[2]

```
template             = template_decl
```

2. 为消除 < 和 > 作为关系运算符或模板名分隔符的歧义性, 一旦一个 structure_name 或者 procedure_name 在语法分析时被作为一个 template 的一部分, 它就变成了一个终结符号.

```
                    (structure | procedure | specialization).
specialization = "struct" structure_name "<" additive_list ">"
                    [structure_body] ";".
template_decl  = "template" "<" [parameter_list] ">" [constraint].
constraint     = "requires" "(" expression ")".

template_name  = (structure_name | procedure_name)
                    ["<" additive_list ">"].
additive_list  = additive {"," additive}.
```

当一个 template_name 被用作原语时, 相应的模板定义将用于生成实际的结构或过程, 其中的模板参数都用对应的模板实参取代. 这些模板实参或者是显式提供, 放在 template_name 之后作为其实参, 或者 (对于过程而言) 是根据过程的参数类型推导出来.

模板结构可以专门化, 以提供了该模板的另一定义. 当其参数的匹配优于对应的未专门化的模板版本时, 就会考虑它的使用.

当模板定义包含一个约束 (constraint) 部分, 模板的参数类型和值都必须满足 requires 后面的布尔表达式.

内部定义

pointer(T) 是一个类型构造符, 返回指向 T 的指针类型. 如果 x 是类型 T 的对象, addressof(x) 返回一个 pointer(T) 类型的值, 也就是对 x 的引用. source、sink 和 deref 都是定义在指针类型上的一元函数. source 对所有指针类型有定义, 返回对应的常量引用, 参看第 6.1 节. sink 和 deref 仅对指向非常量对象的指针类型有定义, 返回对应的非常量引用, 参看第 9.1 节. reinterpret_cast 是一个函数模板, 它以一个引用类型和一个对象 (以引用方式传递) 为参数, 返回指向该对象的具有特定的引用类型的引用. 该对象必须存在基于这一引用类型的解释.

B.2 宏和特征结构

为使 B.1 节定义的语言能作为合法的 C++ 程序进行编译, 需要有几个宏和结构定义.

模板约束

requires 子句用下面的宏实现:[3]

```
#define requires(...)
```

内部定义

引入 pointer(T) 和 addressof(x) 是为了给出一种线性的描述方式, 而且支持简单的自上而下语法分析. 它们的实现如下

```
#define pointer(T) T*


template<typename T>
pointer(T) addressof(T& x)
{
    return &x;
}
```

类型函数

类型函数通过一种称为特征类 (trait class) 的 C++ 技术实现. 对每个类型函数, 例如 ValueType, 我们都定义一个结构模板, 这里是 value_type<T>. 该结构模板只包含一个 typedef, 为方便起见这里用名字 type; 如果适宜, 可以在基结构模板里提供一个默认的定义:

```
template<typename T>
struct value_type
{
```

3. 这一实现意味着只把需求看作文档.

```
    typedef T type;
};
```

为了提供一种方便的使用方式, 我们定义一个宏[4], 它提取出相应的 typedef 作为类型函数作用的结果:

```
#define ValueType(T) typename value_type< T >::type
```

这里为特点类型做一个专门化, 作为全局定义的精化:

```
template<typename T>
struct value_type<pointer(T)>
{
    typedef T type;
};
```

4. 这样的宏只在模板定义里可用, 因为其中用了关键词 typename.

参考文献

Saurabh Agarwal and Gudmund Skovbjerg Frandsen. Binary GCD like algorithms for some complex quadratic rings. In Duncan A. Buell, editor, *Algorithmic Number Theory,* 6th International Symposium, Burlington, VT, USA, June 13–18, 2004. *Proceedings*, volume 3076 of *Lecture Notes in Computer Science*, pages 57–71. Springer, 2004.

Jon Bentley. Programming pearls. *Communications of the ACM*, 27(4):287–291, 1984.

Bernard Bolzano. *Rein analytischer Beweis des Lehrsatzes, daß zwischen je zwey Werthen, die ein entgegengesetztes Resultat gewhren, wenigstens eine reelle Wurzel der Gleichung liege.* Gottlieb Haase, Prague, 1817.

Raymond T. Boute. The Euclidean definition of the functions div and mod. *ACM Transactions on Programming Languages and Systems*, 14(2):127–144, April 1992.

Robert S. Boyer and J Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, October 1977.

Richard P. Brent. An improved Monte Carlo factorization algorithm. *BIT*, 20:176–184, 1980.

Augustin-Louis Cauchy. *Cours D'Analyse de L'Ecole Royale Polytechnique.* L'Académie des Sciences, 1821.

G. Chrystal. *Algebra: An Elementary Text-Book. Parts I and II.* Adam and Charles Black, 1904. Reprint, AMS Chelsea Publishing, 1964.

James C. Dehnert and Alexander A. Stepanov. Fundamentals of generic programming. In Mehdi Jazayeri, Rüdiger G. K. Loos, and David R. Musser, editors, *Generic Programming,* International Seminar on Generic Programming, Dagstuhl Castle, Germany, April/May 1998. *Selected Papers*, volume 1766 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 2000.

Persi Diaconis and Paul Erdös. On the distribution of the greatest common divisor. In Anirban DasGupta, editor, *A Festschrift for Herman Rubin*, volume 45 of *Lecture Notes—Monograph Series*, pages 56–61. Institute of Mathematical Statistics, 2004.

Edsger W. Dijkstra. Notes on structured programming. In O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*, pages 1–82. Academic Press, London and New York, 1972.

P. G. L. Dirichlet. *Forlesungen über Zahlentheorie.* Vieweg und Sohn, 1863. With supplements by Richard Dedekind. English translation by John Stillwell. *Lectures on Number Theory*, American Mathematical Society and London Mathematical Society, 1999.

Krzysztof Dudziński and Andrzej Dydek. On a stable minimum storage merging algorithm. *Information Processing Letters*, 12(1):5–8, February 1981.

Barry Dwyer. Simple algorithms for traversing a tree without an auxiliary stack. *Information Processing Letters*, 2:143–145, 1974.

Charles M. Fiduccia. An efficient formula for linear recurrences. *SIAM Journal on Computing*, 14(1):106–112, February 1985.

William Fletcher and Roland Silver. Algorithm 284: Interchange of two blocks of data. *Communications of the ACM*, 9(5):326, May 1966.

Robert W. Floyd and Donald E. Knuth. Addition machines. *SIAM Journal on Computing*, 19(2):329–340, 1990.

Georg Ferdinand Frobenius. Über endliche gruppen. In *Sitzungesberichte der Königlich Preussischen Akademie der Wissenschaften zu Berlin*, Phys.-math. Classe, pages 163–194, Berlin, 1895.

Hermann Günther Grassmann. *Lehrbuch der Mathematik für höhere Lehranstalten*, volume 1. Enslin, Berlin, 1861.

David Gries and Harlan Mills. Swapping sections. Technical Report 81-452, Department of Computer Science, Cornell University, January 1981.

Sir Thomas L. Heath. *The Thirteen Books of Euclid's Elements.* Cambridge University Press, 1925. Reprint, Dover, 1956.

T. L. Heath. *The Works of Archimedes.* Cambridge University Press, 1912. Reprint, Dover, 2002.

C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

Kenneth Iverson. *A Programming Language.* Wiley, 1962.

Donald E. Knuth. *The Art of Computer Programming* Volume 1, fascicle 1: *MMIX: A RISC Computer for the New Millenium.* Addison-Wesley, Boston, 2005.

Donald E. Knuth. *The Art of Computer Programming* Volume 2: *Seminumerical Algorithms* (3rd edition). Addison-Wesley, Reading, MA, 1997.

Donald E. Knuth. *The Art of Computer Programming* Volume 3: *Sorting and Searching* (2nd edition). Addison Wesley, Reading, MA, 1998.

Donald E. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977.

Jin Ho Kwak and Sungpyo Hong. *Linear Algebra.* Birkhäuser, 2004.

J.-L. Lagrange. *Leçons élémentaires sur les mathématiques, données à l'école normale en 1795.* 1795. Reprinted: *Oeuvres*, vol. VII, pages 181–288. Paris: Gauthier-Villars, 1877.

Leon S. Levy. An improved list-searching algorithm. *Information Processing Letters*, 15(1): 43–45, August 1982.

Gary Lindstrom. Scanning list structures without stack or tag bits. *Information Processing Letters*, 2:47–51, 1973.

John W. Mauchly. Sorting and collating. In *Theory and Techniques for Design of Electronic Digital Computers.* Moore School of Electrical Engineering, University of Pennsylvania, 1946. Reprinted in: *The Moore School Lectures*, eds. Martin Campbell-Kelly and Michael R. Williams, pages 271–287. Cambridge, Massachusetts: MIT Press, 1985.

D. P. McCarthy. Effect of improved multiplication efficiency on exponentation algorithms derived from addition chains. *Mathematics of Computation*, 46(174):603–608, April 1986.

J. C. P. Miller and D. J. Spencer Brown. An algorithm for evaluation of remote terms in a linear recurrence sequence. *The Computer Journal*, 9(2):188–190, 1966.

Joseph M. Morris. Traversing binary trees simply and cheaply. *Information Processing Letters*, 9(5):197–200, 1979.

David R. Musser. Multivariate polynomial factorization. *Journal of the ACM*, 22(2):291–308, April 1975.

David R. Musser and Gor V. Nishanov. A fast generic sequence matching algorithm. Tech. Rep., Computer Science Department, Rensselaer Polytechnic Institute, 1997. Archived as `http://arxiv.org/abs/0810.0264v1`arXiv:0810.0264v1[cs.DS].

Giuseppe Peano. *Formulario Mathematico, Editio V.* Fratres Bocca Editores, Torino, 1908. Reprinted: Roma: Edizioni Cremonese, 1960.

R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

Gay Robins and Charles Shute. *The Rhind Mathematical Papyrus.* British Museum Publications, 1987.

J. M. Robson. An improved algorithm for traversing binary trees without auxiliary stack. *Information Processing Letters*, 2:12–14, 1973.

H. Schorr and W. M. Waite. An efficient and machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.

Robert Sedgewick, Thomas G. Szymanski, and Andrew C. Yao. The complexity of finding cycles in periodic functions. *SIAM Journal on Computing*, 11(2):376–390, 1982.

Laurence E. Sigler. *Fibonacci's Liber Abaci: Leonardo Pisano's Book of Calculation.* Springer-Verlag, 2002.

Josef Stein. Computational problems associated with Racah algebra. *J. Comput. Phys.*, 1: 397–405, 1967.

Alexander Stepanov and Meng Lee. The Standard Template Library. Technical Report 95-11(R.1), HP Laboratories, November 1995.

Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

Ken Thompson and Dennis Ritchie. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, 1974.

Bartel Leenert van der Waerden. *Moderne Algebra* Erster Teil. Julius Springer, 1930. English translation by Fred Blum. *Modern Algebra*, New York: Frederic Ungar Publishing, 1949.

André Weilert. (1+i)-ary GCD computation in $\mathbb{Z}[i]$ as an analogue of the binary GCD algorithm. *J. Symb. Comput.*, 30(5):605–617, 2000.

Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11):822–823, 1977.

索　引

符号索引

英文索引

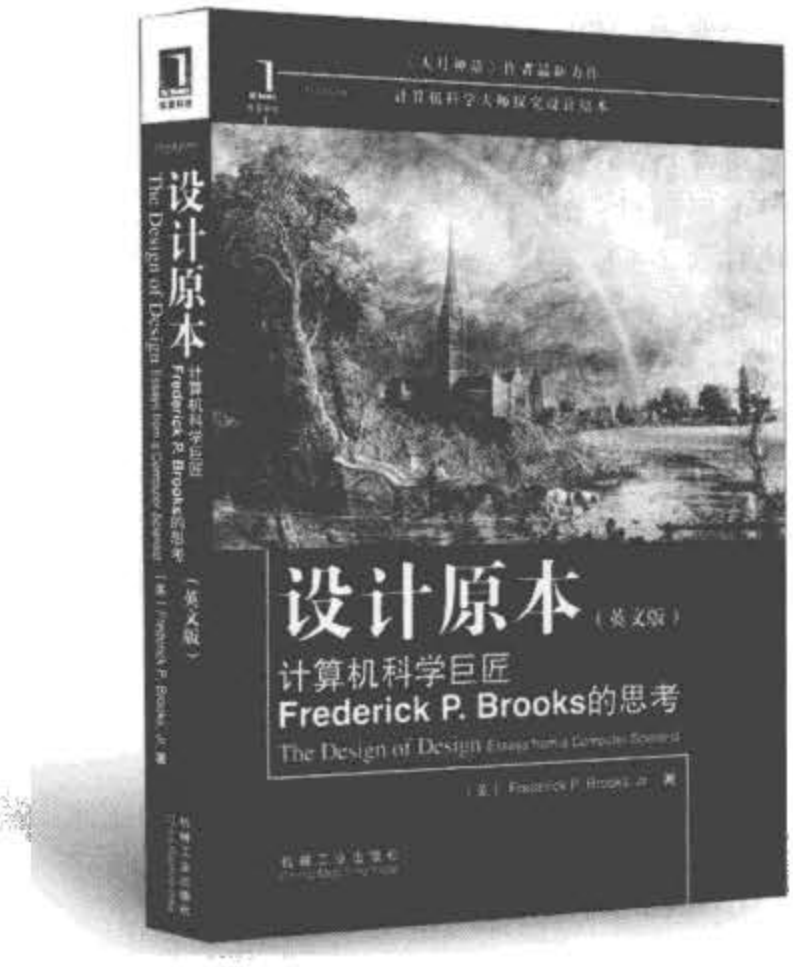
A

C

Q

中文索引

ISBN：978-7-111-32557-4
定价：55.00



ISBN：978-7-111-32503-1
定价：69.00

《人月神话》作者最新力作　计算机科学大师探究设计原本

本书包含了多个行业设计者的特别领悟。Frederick P. Brooks, Jr.精确发现了所有设计项目中内在的不变因素，揭示了进行优秀设计的过程和模式。通过与几十位优秀设计者的对话，以及他自己在几个设计领域的经验，作者指出，大胆的设计决定会产生更好的结果。

本书几乎涵盖所有有关设计的议题：从设计哲学谈到设计实践，从设计过程到设计灵感，既强调了设计思想的重要性，又对沟通中的种种细节都做了细致入微的描述，并且谈到了因地制宜做出妥协的具体准则。

深入理解计算机系统（原书第2版）
ISBN：978-7-111-32133-0
定价：99.00

深入理解计算机系统（英文版·第2版）
ISBN：978-7-111-32631-1
定价：128.00

Linux内核设计与实现（英文版·第3版）
ISBN：978-7-111-32792-9
定价：69.00

Linux内核设计与实现（原书第3版）
ISBN：978-7-111-33829-1
定价：69.00元

www.hzbook.com

填写读者调查表　加入华章书友会
获赠精彩技术书　参与活动和抽奖

尊敬的读者：

感谢您选择华章图书。为了聆听您的意见，以便我们能够为您提供更优秀的图书产品，敬请您抽出宝贵的时间填写本表，并按底部的地址邮寄给我们（您也可通过www.hzbook.com填写本表）。您将加入我们的“华章书友会”，及时获得新书资讯，免费参加书友会活动。我们将定期选出若干名热心读者，免费赠送我们出版的图书。请一定填写书名书号并留全您的联系信息，以便我们联络您，谢谢！

书名：　　书号：7-111-(　　)

姓名：	性别：☐男　☐女	年龄：	职业：
通信地址：		E-mail：	
电话：	手机：	邮编：	

1. 您是如何获知本书的：

☐朋友推荐　☐书店　☐图书目录　☐杂志、报纸、网络等　☐其他

2. 您从哪里购买本书：

☐新华书店　☐计算机专业书店　☐网上书店　☐其他

3. 您对本书的评价是：

技术内容	☐很好	☐一般	☐较差	☐理由_____
文字质量	☐很好	☐一般	☐较差	☐理由_____
版式封面	☐很好	☐一般	☐较差	☐理由_____
印装质量	☐很好	☐一般	☐较差	☐理由_____
图书定价	☐太高	☐合适	☐较低	☐理由_____

4. 您希望我们的图书在哪些方面进行改进？

_____

_____

5. 您最希望我们出版哪方面的图书？如果有英文版请写出书名。

_____

_____

6. 您有没有写作或翻译技术图书的想法？

☐是，我的计划是_____　☐否

7. 您希望获取图书信息的形式：

☐邮件　☐信函　☐短信　☐其他_____

请寄：北京市西城区百万庄南街1号　机械工业出版社　华章公司　计算机图书策划部收

邮编：100037　电话：（010）88379512　传真：（010）68311602　E-mail: hzjsj@hzbook.com

编程原本

Elements of Programming

"要是问一位机械、建筑或电子工程师，如果不依靠坚实的数学基础，他们能走多远。他们会告诉你'走不了多远'。而所谓的软件工程师在实践其技能时，却常常对他们所做工作的数学基础知之甚少，甚至一无所知。同时我们也很奇怪为什么软件由于不能按时发布并充斥错误而声名狼藉，而其他工程师却能按时完成其桥梁、汽车、各种电子装置等，而且缺陷很少。本书就是想纠正这种不平衡现象。我在Adobe的高级开发团队的成员们，但凡参加了基于同样材料的课程，都觉得付出的时间获益匪浅。初看可能觉得这种高度技术性的文字只是为计算机科学家写的，其实所有从事实际工作的软件工程师都应该来读。"

—— Martin Newell，Adobe 院士

"本书包含一些我所见过的最美的代码。"

—— Bjarne Stroustrup，C++ 设计者

"我很高兴看到Alex课程的内容。担任Silicon Graphics的CTO时，我曾大力支持这一课程的开发和教授。现在这本书已经能被所有程序员阅读了。"

—— Forest Baskett，合伙人，New Enterprise Associates

"Paul的耐心和在体系结构方面的经验帮助把Alex的数学方法组织成为一套高度结构化的大质——功德无量！"

—— Robert W. Taylor，Xerox PARC CSL和DEC系统研究中心创始人

书号：978-7-111-30027-4
定价：49.00元

上架指导：计算机 程序设计
ISBN 978-7-111-36729-
9 787111 367291
定价：59.00元

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com
华章网站 http://www.hzbook.com
网上购书：www.china-pub.com

PEARSON
www.pearson.com